Dokumentation

Universität Paderborn

Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik
Fachgruppe: Algorithmen und Komplexität

Heinz Nixdorf Institut
Institut für Informatik
Fürstenallee 11
33102 Paderborn

Prof. Dr. math. Friedhelm Meyer auf der Heide

Raum: F1.301
Telefon: +49 (0) 5251-60-6480
Telefax: +49 (0) 5251-60-6482
eMail: fmadh@upb.de

Paderborn, 01.10.2007

Inhaltsverzeichnis

1. Einführung **1**
- 1.1. Motivation . . . 1
- 1.2. Ziele . . . 1
- 1.3. Modell . . . 2
- 1.4. Projektteam . . . 3

2. Architektur und Komponenten **5**
- 2.1. Architektur des Gesamtsystems . . . 5
- 2.2. Simulationskernel . . . 5
- 2.3. Robotermodul . . . 5
- 2.4. Visualisierung . . . 6
- 2.5. Editor . . . 6
- 2.6. Messgrößen . . . 6

3. Simulationskernel **7**
- 3.1. Architektur . . . 7
- 3.2. Simulationsabwicklung und -verwaltung . . . 8
  - 3.2.1. Multithreading . . . 8
  - 3.2.2. Messages . . . 9
  - 3.2.3. Roboterverwaltung . . . 11
- 3.3. Datenstrukturen . . . 16
  - 3.3.1. Quadtree für Speicherung der Map . . . 16
  - 3.3.2. PRQuadtree für Speicherung der Schätze . . . 19
- 3.4. RMI . . . 21

4. Roboter-Modul **22**
- 4.1. Architektur . . . 22
  - 4.1.1. RobotControl . . . 24
  - 4.1.2. PhysicalUnit . . . 24
  - 4.1.3. RobotData und Wegberechnung . . . 25
- 4.2. Strategie- und Taktikkonzept . . . 29
  - 4.2.1. Strategiekonzept . . . 29
  - 4.2.2. Taktikkonzept . . . 31
  - 4.2.3. Schnittstellen für die Strategieentwicklung . . . 33
- 4.3. Taktiken . . . 40
  - 4.3.1. CollisionDetector . . . 40
  - 4.3.2. HeadForLocations . . . 44
  - 4.3.3. HeadForRobots . . . 45
  - 4.3.4. TeamManagement . . . 46
  - 4.3.5. SecureEnergyManagement . . . 52
- 4.4. Strategien . . . 53
  - 4.4.1. DividedExplorationStrategy . . . 53
  - 4.4.2. SubAreaStrategy . . . 55
  - 4.4.3. ExplorationGreedStrategy . . . 59
  - 4.4.4. TreasureGreedStartegy . . . 59

4.4.5. WorkerStrategy . . . 61
4.4.6. TransporterStrategy . . . 62

5. Visualisierung **64**
5.1. Architektur . . . 64
5.2. Wichtige Klassen . . . 65
5.2.1. Anbindung an den Kernel und Ablaufsteuerung . . . 66
5.2.2. Datenhaltung . . . 67
5.2.3. 2D-Visualisierung . . . 69
5.2.4. Detailansicht . . . 70
5.2.5. Erkundete Karte und abgelaufene Wege . . . 70
5.2.6. 3D-Visualisierung . . . 70
5.3. Preprocessing . . . 73
5.3.1. 2D-Gelände . . . 73
5.3.2. 3D-Gelände . . . 73
5.3.3. Ein paar Zahlen... . . . 80

6. Editor **81**
6.1. Architektur . . . 81
6.1.1. Die Bigmap . . . 81
6.1.2. Die Minimap . . . 82
6.1.3. Die Optionspalette . . . 82
6.1.4. Die Menüleiste . . . 83
6.1.5. Die Statuszeile . . . 84
6.2. Schätzeverwaltung . . . 84
6.3. Roboterkonfiguration . . . 84
6.4. Datenstruktur . . . 86
6.5. Generator . . . 87
6.5.1. Architektur . . . 87
6.5.2. Wizard Framework . . . 87
6.5.3. Generatorimplementierung . . . 89

7. Messgrößen **91**
7.1. Einleitung . . . 91
7.2. Ablauf und Architektur . . . 91
7.3. Sammeln der Messwerte . . . 92
7.4. Benutzereigene Messgrößen . . . 93

8. Ausblick/Offene Punkte **94**

A. Projekt in Eclipse einbinden **95**

1. Einführung

Die Projektgruppe *Smart Teams: Local, Distributed Strategies for Self-Organizing Robotic Exploration Teams* fand vom Wintersemester 2007/08 bis zum Wintersemester 2008/09 statt. Sie hatte das Ziel einen Software-Simulator für einen Schwarm von Robotern zu entwickeln. In diesem Kapitel wird das Thema zunächst motiviert und danach die Ziele für das Projekt vorgestellt. Darauf aufbauend wird nachfolgend ein formales Modell beschrieben. Zuletzt findet sich noch eine Auflistung des Projektteams.

1.1. Motivation

Stellen wir uns ein Szenario vor, in dem ein Team von Explorationsrobotern – wir nennen es ein *Smart Team* – sich selbst organisieren muss um ein unbekanntes Terrain zu erkunden und darin bestimmte Aufgaben zu erledigen. Das Thema ist von großem wissenschaftlichen und wirtschaftlichen Interesse. Praktische Anwendungen sind z.B. Rettungseinsätze in einem schwer zugänglichen Gelände, sowie Expeditionen im Ozean oder auf einem fremden Planeten. Zur Entwicklung von lokalen, verteilten Algorithmen zu Steuerung des Roboterverhaltens kann ein Software-Simulator eine große Hilfe sein. Beim Entwurf eines solchen Simulators gibt es zwei wesentliche Herausforderungen. Die erste Herausforderung ist, dass Roboter nur beschränktes lokales Wissen über den globalen Status des Systems haben. Daher muss der Simulator für jeden Roboter eine individuaelle, abstrahierte Sicht des Terrains anbieten. Angenommen in einem Terrain wird jedes Hindernis durch vier 32-bit *float* Werte (x-/y-Koordinate, Länge, Breite) gespeichert. Bei 1 Millionen Hindernisen und 1000 Robotern würde dies bereits ein Datenvolumen von ca. 15 GB nach sich ziehen, das durchgängig im Arbeitsspeicher gehalten werden müsste. Moderne Simulatoren sind nicht fähig diese Anforderungen zu erfüllen. Es werden besonders platzsparende Datenstrukturen benötigt, die effiziente Anfragen und Aktualisierungen erlauben. Die zweite Herausforderung liegt in der Möglichkeit, Strategien besonders schnell Entwickeln zu können (*Rapid Prototyping*). Dies kann durch die Bereitstellung einer einfachen Schnittstelle zum Einbinden von Strategien realisiert werden, die ein einfachs Framework bereitstellt um Routineaufgaben zu erledigen.

1.2. Ziele

Für das vorangehende Szenario und die Roboter ergaben sich vier primäre Ziele:

- *EXPLORATION:* Es sollen Methoden zur Exploration von unbekanntem Gelände entwickelt und implementiert werden. Diese sollen gewährleisten, dass das gesamte Gelände schnell nach Schätzen durchgesucht wird. Dabei sind Hindernisse zu umzugehen.
- *SCHEDULING:* Es sollen Methoden zur "Bearbeitung" der Schätze entwickelt und implementiert werden. Für das Ausgraben und Transportieren der Schätze müssen ensprechend qualifizierte Roboter eingesetzt werden

(ein Bagger ist ja gleichzeitig nicht LKW). Das heisst, dass Roboter gerecht eingeteilt werden müssen, damit die anliegende Arbeit schnell erledigt ist.

- *COMMUNICATION:* Damit die Roboter sich effektiv organisieren können, muss ein Kommunikationsmedium verfügbar sein. Es sollen Methoden entwickelt und umgesetzt werden, die eine Kommunikation zwischen den Robotern gewährleisten, auch wenn sich diese in grosser Entfernung befinden. Das setzt voraus, das mobile Relay-Stationen an bestimmten Stellen eingefügt werden und Funksignale weiterleiten.
- *3D-VISUALISATION:* Die Bemühungen der Roboter im Gelände sollen in 3D-Grafik den Interessenten vorgestellt werden. Es entsteht Bedarf für eine schöne Echtzeit-Visualisierung.

All diese Strategien werden *nicht* zentral koordiniert – die Roboter sind selber für die Organisation verantwortlich. Damit das Kommunikationsnetzwerk und die Computer der Roboter nicht mit Daten überlastet werden, sollten entwickelte Strategien effizient verteilt und lokal Arbeiten, d.h. mit einer beschränkten Sicht auf den Zustand des Systems auskommen. Die so erarbeiteten Lösungen werden nicht unbedingt optimal sein, dafür können die Roboter autonom agieren und können auch im Fall des Ausfalls Ihrer Kollegen weitermachen.
Es war gefordert ein komplettes System zu erstellen, welches sich aus den dargestellten Komponenten zusammensetzt. Es war nicht nur Softwaretechnische Arbeit am System gefordert, sondern auch konzeptionelle Mitwirkung an der Entstehung der Strategien.

1.3. Modell

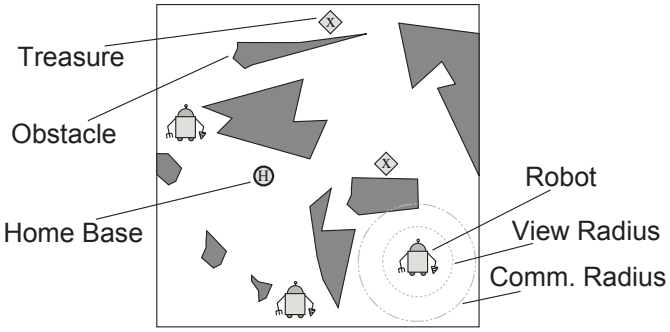


Abbildung 1: Illustration des Modells

Die wesentlichen Elemente des Modells sind in Abbildung 1 dargestellt. Die Karte auf der die Roboter agieren nehmen wir als einen zweidimensionalen, rechteckigen Ausschnitt aus einer Ebene an. Darauf befinden sich eine Vielzahl von Hindernissen mit beliebiger Form. Sie definieren Areale in denen Roboter sich nicht bewegen dürfen. Es gibt zusätzlich eine einzelne Basisstation welche als Startpunkt für alle Roboter dient. Dieses sind die statischen Elemente, welche sich nach dem Start einer Simulation nicht mehr verändern.

Im Gegensatz dazu gibt es noch dynamische Elemente: Roboter und Schätze. Letztere können von Roboter ausgegraben und transportiert werden. Sie können an jeder beliebigen stelle auf der Karte in unterschiedlichen Quantitäten vorkommen. Desweiteren können sich die Roboter in den passierbaren Regionen völlig frei bewegen. Jeder Roboter hat einen Energievorrat der sich im Laufe der Zeit verringert (abhängig von den Aktionen die er ausführt), aber an der Basisstation wieder aufgeladen werden kann. Sollte ein Roboter jemals seinen kompletten Energievorrat verbrauchen, so ist er für den Rest der Simulation deaktiviert. Roboter empfangen Informationen über ihre Umgebung innerhalb eines festen, aber frei wählbaren Sichtradius. Kommunikation mit anderen Robotern und somit der Austausch von Informationen ist analog durch einen Funkradius beschränkt. Es gibt verschiedene Robotertypen die alle gewisse Restriktionen haben. Beispielsweise kann ein Transporter keine Schätze ausgraben, wohl aber welche transportieren. Jeder Roboter muss Informationen über seine Umgebung (Hindernisse, Roboter, Schätze) lokal ermitteln und seine eigene (abstrakte) Repräsentation davon erstellen. Geteiltes bzw. gemeinsames Wissen kann nur durch den expliziten Austausch von Information über Kommunikation geschehen. Alle Roboter handeln völlig autonom, d.h. jeder Roboter hat eine eigene Strategie von der sein Verhalten gesteuert wird. Das Hauptziel der Roboter als Kollektiv ist, die Karte vollständig zu explorieren, alle Schätze auszugraben und sie anschliessend zur Basisstation zu bringen. Die Roboter können dabei unterschiedliche Teilaufgaben übernehmen, je nach Robotertyp und Strategie.

1.4. Projektteam

Teilnehmer der Projektgruppe:

- Stephan Arens
- Alexander Buss
- Helena Deck
- Holger Hagedorn
- Peter Isaak
- Alexander Krieger
- Viktor Nesterow
- Adrian Ogierman
- Jonas Schrieb
- Boris Stobbe
- Thomas Storm
- Henning Wachsmuth

Projektleiter: Holger Hagedorn

Betreuer der AG Meyer auf der Heide:

- Prof. Dr. Friedhelm Meyer auf der Heide
- Dr. Matthias Fischer
- Miroslaw Dynia
- Jaroslaw Kutylowski

2. Architektur und Komponenten

In diesem Abschnitt wird kurz vorgestellt, aus welchen einzelnen Modulen das System zusammengesetzt und wofür diese Teilmodule jeweils verantwortlich sind. Eine genauere Analyse der Teilmodule findet sich in ihren jeweiligen eigenen Kapiteln.

2.1. Architektur des Gesamtsystems

Abbildung 2 zeigt die grobe Architektur des gesamten Smartteams-Systems.

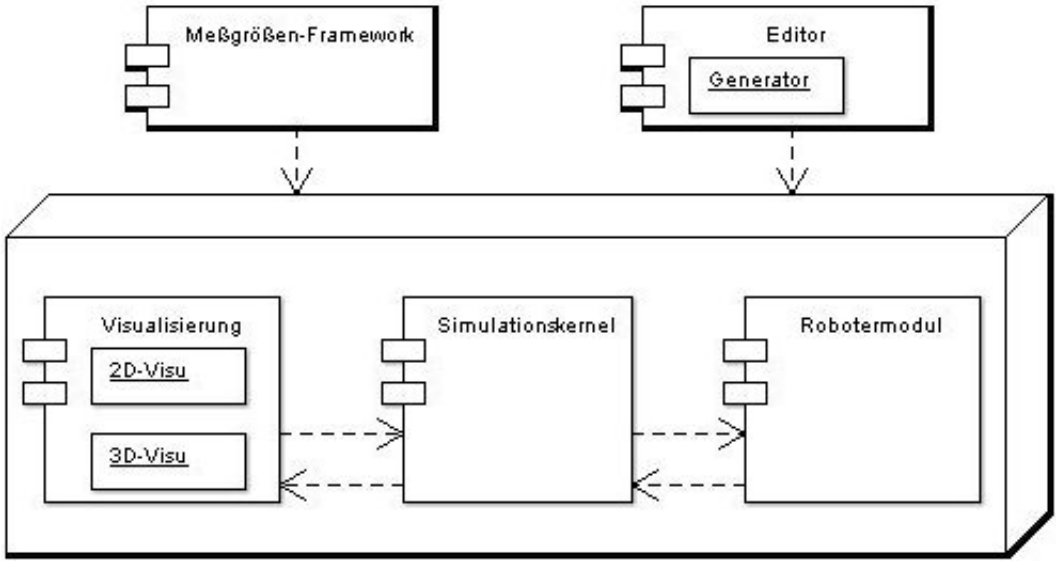


Abbildung 2: Die Architektur des Gesamtsystems

Das System setzt sich dabei aus den Hauptkomponenten Simulationskernel, Robotermodul und Visualisierung zusammen; zusätzlich umfaßt es einen Editor und das Meßgrößen-Framework. Diese Komponenten werden im folgenden kurz erläutert und in ihren jeweiligen Kapiteln detailliert dargestellt.

2.2. Simulationskernel

Im **Simulationskernel** läuft die eigentliche Simulation ab. Er speichert die Karte und führt Roboteraktionen auf ihr durch. Des weiteren wickelt er die Kommunikation unter den Robotern ab und bietet eine Schnittstelle für die Visualisierung der Simulation. Eine genaue Beschreibung findet sich in Kapitel 3.

2.3. Robotermodul

Das **Robotermodul**, detailliert beschrieben in Abschnitt 4, ist über Schnittstellen mit dem Simulationskernel verbunden und dafür verantwortlich, die Roboteraktionen zu berechnen und vorauszuplanen. Des Weiteren ist es für die

Datenhaltung der Roboter verantwortlich. **Strategien** und **Taktiken** werden im Robotermodul implementiert.

2.4. Visualisierung

Die Visualisierung (ausführliche Beschreibung in Abschnitt 5) teilt sich aus Benutzersicht in zwei Teile auf: Die zwei- und die dreidimensionale Visualisierung. Die zweidimensionale Visualisierung ist ideal zur Evaluierung von Strategien und Taktiken geeignet – der Benutzer kann sich dort viele Details anzeigen lassen, beispielsweise den bereits explorierten Teil der Karte. Die dreidimensionale Visualisierung bietet eine attraktive Sicht auf die Roboter und die Karte.

2.5. Editor

Der Editor ist ein unerläßlicher Teil des Gesamtprojektes: Mit diesem Werkzeug kann man komfortabel Karten, Roboter- und Bodenobjektkonfigurationen erstellen und bearbeiten. Daran angebunden ist ein Generator, der Karten mittels vieler verschiedener Parameter automatisch generiert oder sie aus Bilddateien ausliest und in ein für den Simulator lesbares Format transferiert. Mehr über Features und zur Funktionsweise findet sich in Abschnitt 6.

2.6. Messgrößen

Das Meßgrößen-Framework ist ein Analysetool für das Smartteams-Projekt. Mittels diesem Framework kann man eine reichhaltige Auswahl an Daten, beispielsweise über Roboterbewegungen, aufzeichnen und sich die Ergebnisse in der Form verschiedenartiger Graphen darstellen lassen. Das ermöglicht eine umfassende Analyse verwendeter Strategien und Taktiken. Dokumentiert ist es in Abschnitt 7.

3. Simulationskernel

Dieses Kapitel beschreibt den Simulationskernel. Zunächst wird ein kurzer Überblick über die Architektur gegeben, im weiteren Verlauf dann werden Konzepte und Datenstrukturen des Kernels detailliert beschrieben.

3.1. Architektur

Ein Überblick über die Architektur des Simulationskernels findet sich in Abbildung 3.

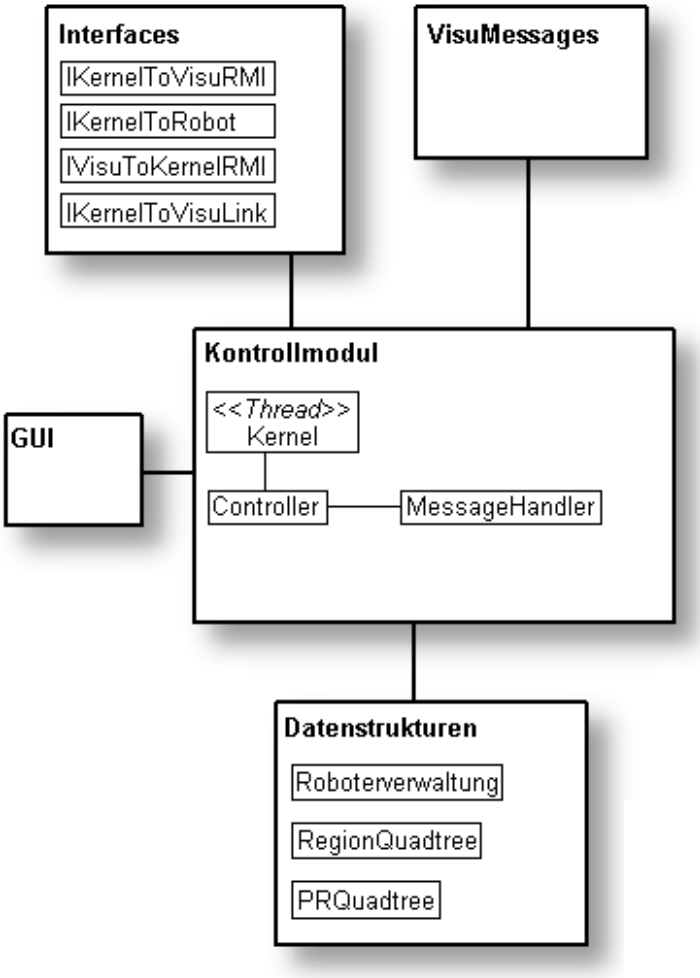


Abbildung 3: Die Architektur des Simulationskernels

Daraus ist zu erkennen, dass der Hauptteil der Simulation durch das **Kontrollmodul** verwaltet wird. Hier laufen die einzelnen Simulationsschritte ab, werden die Nachrichten verwaltet (sowohl die Roboternachrichten als auch die RMI-Nachrichten zur Visualisierung), hier hat die Roboterverwaltung ihren Sitz und vieles mehr. Die wichtigsten Klassen dieses Moduls sind `Controller` und

`Kernel` sowie `MessageHandler` und `RobotObjects`. Die Konzepte, die in diesem Modul Anwendung finden, sind in Abschnitt 3.2 ausführlich beschrieben.
Für die Simulation unerläßlich sind des Kernels Datenstrukturen. Diese werden in ihrem eigenen Abschnitt 3.3 ausführlich behandelt. Auch sie sind mit dem Kontrollmodul verbunden.
Die **Interfaces** dienen der Kommunikation: Die Kommunikation mit dem Robotermodul läuft dabei über das Interface `IKernelToRobot`; dieses wird im Kontrollmodul implementiert. Die Kommunikation mit der Visualisierung ist an zweierlei Stellen implementiert: Einerseits mittels Interfaces, die ebenfalls vom Kontrollmodul implementiert werden und die grundsätzliche Kommunikation über RMI ermöglichen und andererseits über eine ganze Reihe von **VisuMessages**. Diese kapseln jeweils wichtige Informationen, beispielsweise über Simulationsstatus oder über Roboteraktionen, und können sowohl von Kernel zur Visualisierung (für beispielsweise Roboterdetails), als auch von der Visualisierung zum Kernel (beispielsweise zur entfernten Steuerung des Kernels) gesendet werden. Mehr dazu findet sich in Abschnitt 3.2.2.
Zu guter Letzt gibt es die graphische Benutzerschnittstelle des Kernels; mittels dieser können Karten oder Roboter- und Schatzkonfigurationen geladen werden und die Simulation gestartet und gesteuert werden.

3.2. Simulationsabwicklung und -verwaltung

Das Design des Kernels beschränkt sich auf die Idee von wenigen Teilmodulen. Innerhalb derer lassen sich abermals Submodule identifizieren bzw. Strukturen, welche solchen entsprechen.

Die Hauptarbeit, das Simulieren von Roboterverhalten, verrichtet der Kernel mit Hilfe von Arbeitsthreads. Diese sind zum Abarbeiten der einzelnen Roboter da. Aufgrund der Dual-Kern-Architektur unseres Referenzsystems wurden zwei Arbeitsthreads gewählt, auf welche ich später näher eingehen werde.

Darüber hinaus ist der Kernel die zentrale Einheit zur Nachrichtenabwicklung. Darunter fallen sowohl Nachrichten von Roboter zu Roboter, als auch Nachrichten zu einer oder mehrer Visualisierungseinheit(en).

3.2.1. Multithreading

Das Multithreadingkonzept ist ein Schlüsselkonzept in unserem Simulationsprogramm. Letztlich ist dieses für eine angemessene Simulationsgeschwindigkeit verantwortlich, denn es erlaubt eine vollständige Auslastung eines Dual-Kern-Prozessors.
Benutzt wird es innerhalb der sequentiellen Abarbeitungsreihenfolge des Kernels und parallelisiert die gesamten Roboteraktionen. Um solch ein System nutzen zu können ist es wichtig in der Ladephase der Simulation die Robotereinheiten entsprechend in, in unserem Falle bislang zwei, Teilpackete aufzuteilen. Diese Zuordnung wird einmalig von der jeweiligen Roboterverwaltung getätigt. Hier muss sichergestellt werden, dass sich dies nicht mehr ändert, ansonsten

können Fehler durch gleichzeitigen Mehrfachzugriff auf eine Datenstruktur auftretten.

Bei der Umsetzung wurden Konditionen für die Threads formuliert, um einen verklemmungsfreien und problemlosen Ablauf sicherzustellen. Dabei folgen die Arbeitsthreads folgender Kondition:

$$\textit{if mutex} == 2\textit{, then notifyall, else if mutex} < 2\textit{, then wait}$$

Der Hauptthread hingegen, aus welchem die Arbeitsthreads immer gestartet werden, folgt lediglich der Kondition:

$$\textit{if mutex} < 2 \textit{ wait}$$

Es ist leicht einzusehen, dass hierbei die logische Bedingung lediglich vorsieht solange mit der Abarbeitungsreihenfolge zu warten, bis alle Arbeitsthreads ihre Aufgabe beendet haben.

Die Anpassung des Kernels an das Threadkonzept führte einiges Weitere mit sich. Die zentralen Datenstrukturen mussten nun gegen gleichzeitiges Manipulieren abgesichert werden um Inkonsistenzen und Fehler zu vermeiden. Leider führte der Versuch dies mit *synchronized*-Methoden zu bewerkstelligen zu enormen Performanceeinbrüchen.
Dies veranlasste mich eine Art Pufferstruktur zu etablieren. Genauer bedeutet dies, dass nicht mehr die vom Roboter gewollte Aktion direkt durchgeführt wird (und somit Datenstrukturmanipulationen nach sich zieht), sondern lediglich diese Aktionsausführung als *gewollt* abgespeichert wird. So trägt jeder Roboter seine gewollten Aktionen in eine, für jeden Arbeitsthread existierende, *HashMap* ein. Diese werden anschließend vor Beginn der neuen Runde sequentiell durchlaufen und die zuvor *gewollten* Aktionen nun in die Datenstruktur übernommen.

3.2.2. Messages

Unter *Messages* fassen wir zwei verschiedene Typen auf:

1. Messages zur Kommunikation mit Visualisierungsmodulen
2. Messages zur Inter-Roboter-Kommunikation

Letztere werden im Kapitel **Roboterverwaltung** näher betrachtet.
In diesem Kapitel möchten wir uns mit dem unter Punkt 1 erwähnten Nachrichtentypen beschäftigen. Hierbei ist wichtig zu erkennen, dass *Messages* keine TCP-Packete oder dergleichen sind. Der konkrete Transport wird über das Protokoll *RMI* (vgl. 3.4) abgewickelt. *Messages*, wie wir sie verstehen, behandeln hingegen die generelle logische Information, welche wir zur Kommunikation zwischen Kernel und einem Visualisierungsmodul benutzen.
Aus dem *Message*-Typ, den darin enthaltenen Daten und u.U. der jeweiligen Nachrichtenprotokollphase wird die jeweilige konkrete Handlung abgeleitet.

Folgende *Message*-Typen werden benutzt:

Messagetyp	**Bedeutung**	**Richtung**
ChangeSimulationSpeed	Fordert den Kernel auf die übergebene Mindestrundenzeit zu benutzen.	In
PauseSimulation	Fordert den Kernel auf die Simulation zu pausieren.	In
Register	Fordert den Kernel auf eine Visualisierung zu registrieren und das Nachrichten-protokoll zu initiieren.	In
Unregister	Fordert den Kernel auf eine Visualisierung aus der Registrierung zu löschen.	In/Out
StartDetailPushing	Fordert den Kernel auf Detailinformationen zu bestimmten Objekten zu liefern.	In
StartResumeSimulation	Fordert den Kernel auf die Simulation fortzusetzen.	In
StopDetailPushing	Fordert den Kernel auf bestimmte Detail-informationen nicht mehr zu senden.	In
StopSimulation	Fordert den Kernel auf die Simulatin zu beenden.	In
StartExplorationQuadtree-Pushing	Fordert den Kernel auf das bisher erkundete Gebiet eines Roboters zu übertragen und nach Merges geg. erneut zu übertragen.	In
SendWholeExploration-Quadtree	Fordert den Kernel auf das aktuell erkundete Gebiet aller Roboter zu übertragen.	In
StopExplorationQuadtree-Pushing	Tilgt eine Beobachtung des erkundeten Gebiets übergebener Roboter.	In
TerrainInitialized	Benachrichtigt den Kernel, dass das Terrain erfolgreich bei der Visualisiserung initialisiert wurde.	In
TerrainRequest	Wird aktuell nicht benutzt.	In
Error	Wird aktuell nicht benutzt.	In/Out
Ping	Wird benutzt um die Verbindung zu testen. Eine Pong-Nachricht ist die Antwort.	In/Out
Pong	Wird als Antwort auf eine Ping-Nachricht geschickt.	In/Out
ExplorationQuadtreeUpdate	Sollte ein Merge einer unter Beobachtung stehenden Roboters aufgetretten sein, so wird diese Nachricht geschickt.	Out
WholeExplorationQuadtree	Kapselt den QuadTree aller Roboter.	Out

Tabelle 1: Message-Typen Teil 1

Messagetyp	**Bedeutung**	**Richtung**
InitScene	Kapselt Initialisierungsdaten von Robotern und Bodenobjekten.	Out
InitTerrain	Kapselt den Namen und den Hashcode des zu verwendenden Terrains.	Out
NewRound Runde.	Kapselt Daten aller Änderungen der letzten Out	
GroundObjectDetailUpdate	Kapselt Detailinformationen zu einem Bodenobjekt.	Out
RobotDetailUpdate	Kapselt Detailinformationen zu einem Roboter.	Out
RobotDefect	Sobald ein Roboter ausfällt, wird diese Nachricht geschickt.	Out
SimulationPaused	Benachrichtigt eine Visualisierung, dass die Simulation pausiert wurde.	Out
SimulationSpeedChanged	Benachrichtigt eine Visualisierung, dass die Simulationgeschwindigkeit geändert wurde.	Out
SimulationStopped	Benachrichtigt eine Visualisierung, dass die Simulation beendet wurde.	Out
SimulationResumed	Benachrichtigt eine Visualisierung, dass die Simulation fortgesetzt wurde.	Out
TerrainData	Wird aktuell nicht benutzt.	Out

Tabelle 2: Message-Typen Teil 2

Zu beachten ist hierbei, dass lediglich Nachrichten zum Initialisierungsprozess auch an nicht registrierte Visualisierungen versendet werden. Alle anderen Nachrichten werden nur an registrierte Visualisierungen versenden, einige zudem noch unter zusätzlichen Restriktionen.

In Abbildung 4 ist der vorgeschriebene Nachrichtenverkehr zwischen Kernel und Visualisierung bei Vorhandensein der zu benutzenden Karte auf Visualisierungsseite angezeigt. Ein alternativer Ablauf, welcher derzeitig leider nur vorbereitet aber nicht gänzlich implementiert ist, wird in Abbildung 5 gezeigt.

3.2.3. Roboterverwaltung

Die Roboterverwaltung ist ein Teil der Datenstrukturen vom Kernel. Hier wird alles bezüglich Roboter abgewickelt. Zu den fundamentalen Aufgaben einer Roboterverwaltung zählen:

1. Generierung von Robotern
2. Initialisieren und Pflegen von fundamentalen Datenstrukturen
3. Abwicklung des gesamten interroboter Nachrichtenverkehrs

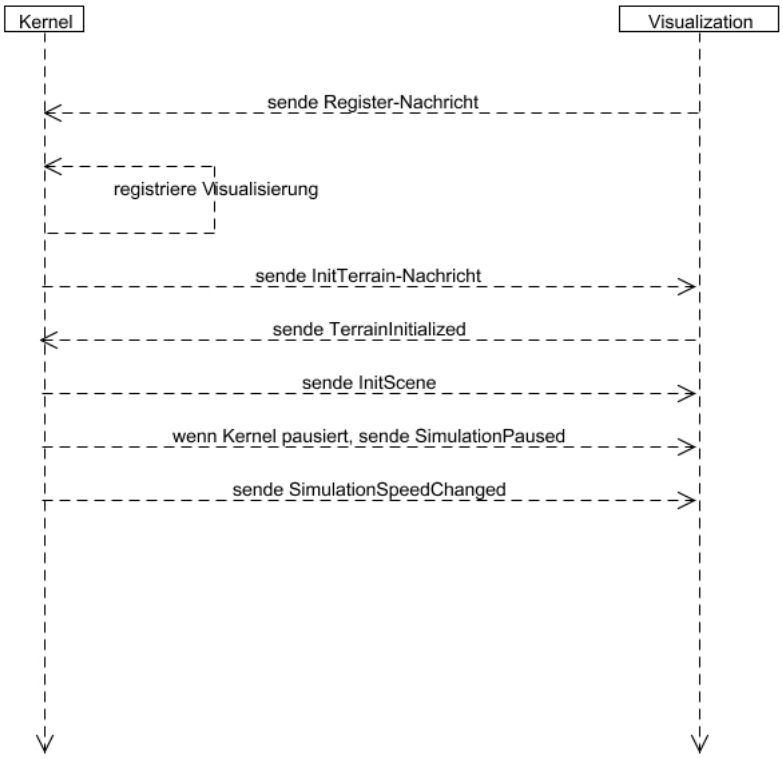


Abbildung 4: Protokoll bei Verfügbarkeit der Karte auf Visualisierungsseite

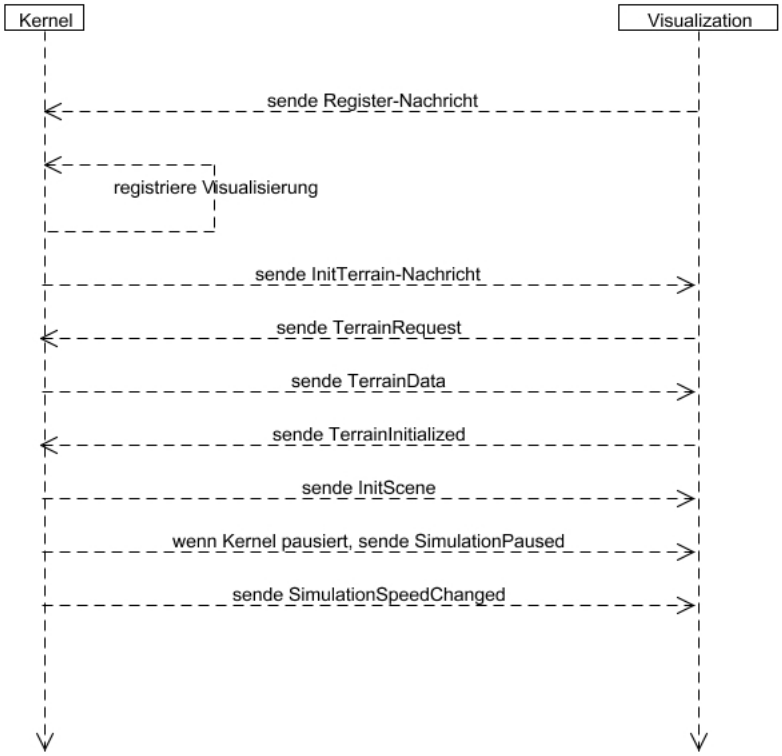


Abbildung 5: Protokoll bei fehlender Karte auf Visualisierungsseite

4. Generierung der RobotInit-Nachricht
5. Sicherstellen einer kontinuierlich gleichbleibenden Ordnung der Roboter

Um eine einheitliche, und somit auch leicht testbare, Basis an Grundeigenschaften sicherzustellen, wurden diese Aufgaben als abstrakte Oberklasse abgelegt. Somit müssen konkrete Roboterverwaltungen sich nur noch, bis auf das Initiieren und Füllen geg. zusätzlich benötigter Strukturen, um die folgenden Elementaraktionen bzw. deren Implementierung kümmern:

1. getRobotsInCommunicationRadius
2. getRobotsInViewRadius
3. moveRobot

Folgende konkrete Roboterverwaltungen existieren im aktuellen System:

1. RobotManagementExact Diese Verwaltung stellt die Grundform dar. Das eigentliche *Bewegen* von Robotern, also die Implementierung von *moveRobot*, ist lediglich ein einfacher Positionswechsel in einem Array.
   Bei den Bereichsanfragen, *getRobotsInCommunicationRadius* und *getRobotsInViewRadius*, werden alle Roboter sequentiell durchlaufen und ein Positionsabgleich durchgeführt, wobei der Radius durch ein Quadrat mit Breite des Radius approximiert wird.
   Wichtig hierbei ist zu erkennen, dass diese Lösung sehr einfach aber bei hoher Roboterzahl nicht mehr performant ist.

2. RobotManagementCellBased Diese Verwaltung stellt eine Verbesserung der Grundform (*RobotManagementExact*) dar. Hierbei wird der Radius immernoch durch ein Quadrat approximiert, jedoch arbeitet diese Verwaltung mit Hilfe einer Zellenaufteilung um performanter zu sein.
   Hierbei bedeutet Zellenaufteilung nichts weiter, als das das Terrain in Schachbrettform in Zellen mit Breite einer Zweierpotenz aufgeteilt wird. Die Zellen sind hierbei quadratisch und von einer Breite, welche immer eine Zweierpotenz sein muss! Letzteres vermeidet Probleme bei der Zellenaufteilung.

   Folgende Aufteilung stellt ein Beispiel dar:

0 ([00,15],[00,15])	1 ([16,31],[00,15])	2 ([33,47],[00,15])	3 ([49,63],[00,15])
4 ([00,15],[16,31])	5 ([16,31],[16,31])	6 ([33,47],[16,31])	7 ([49,63],[16,31])
8 ([00,15],[33,47])	9 ([16,31],[33,47])	10 ([33,47],[33,47])	11 ([49,63],[33,47])
12 ([00,15],[49,63])	13 ([16,31],[49,63])	14 ([33,47],[49,63])	15 ([49,63],[49,63])

Abbildung 6: Beispielaufteilung eines Terrain in Zellen

Die Verwaltung selbst benutzt Zellennummern zur Speicherung, bei der Berechnung in *getRobotsInCommunicationRadius* und *getRobotsInViewRadius* wird hingegen mit Positionen gerechnet (hierzu später mehr).
Die Roboterbewegung ist insoweit angepasst, dass bei einer Umsetzung des Roboters nun die entsprechende Zelle, in welcher der Roboter sich befunden hat, geg. angepasst wird. Verwaltet wird dies mit einer *HashMap* und einem *Array*, wobei ersteres für die Speicherung von Robotern in einer Zelle und letzteres zur Speicherung der Zelle für einen Roboter benutzt wird.

Kommen wir nun zum interessanten Teil, der approximativen Berechnung von Robotern im Sicht- bzw. Kommunikationsradius.
Folgende Schritte werden hierfür in dieser Reihenfolge getätigt:

a) Bestimmte Quadrat des Radius
b) Überprüfe ob Roboter über den Rand seiner aktuellen Zelle blicken kann.
   Dabei wird die Position auf den Mittelpunkt der Zelle approximiert.
   i. Falls ja, so gehe zu Schritt 3
   ii. Falls nein, so gehe zu Schritt 4
c) Überprüfe Roboter im Radius innerhalb der Zelle wie bei RobotManagementExact
d) Erstelle zwei Listen
   i. Liste1 enthällt dabei die äußeren Zellen, d.h. die Zellen, welche vom Radius lediglich geschnitten und nicht gänzlich abgedeckt werden.
   ii. Liste2 enthällt dabei die inneren Zellen, d.h. die Zellen, welche vom Radius gänzlich abgedeckt werden.
e) Berechne alle inneren und äußeren Zellen mit dem Algorithmus *computeDirectNeighbors*.
f) Füge Roboter der inneren Zellen direkt zum Resultat hinzu.
g) Überprüfe Roboter der äußeren Zellen wie in RobotManagementExact.

Algorithmus *computeDirectNeighbors*:

a) Normalisiere[1] eigene Roboterposition
b) Berechne Position der oberen linken Ecke des Quadrats
   i. Korrigiere Randfehler[2] falls notwendig

[1] Positionen innerhalb einer Zelle werden auf den Zellenmittelpunkt normiert.
[2] Durch die Zellenaufteilung bilden sich Randfehler wie folgt: Angenommen wir haben eine Zellenbreite von 16, so gehören Positionen [0,15] zu der jeweiligen Zelle. Jedoch führt die Position 15 zu einer Verzerrung des Quadrat-Radius des Roboters.

c) Berechne linke untere und rechte obere Position

d) Normalisiere alle Eckpositionen

e) Berechne äußere Zellen

   i. Füge erste äußere Zeile hinzu

   ii. Füge ersten und letzten Punkt für weitere Zeilen hinzu

   iii. Füge letzte äußere Zeile hinzu

f) Berechne innere Zellen

   i. Berechne rekursiv äußere Zellen mit geringerem Radius

g) Entferne illegale Positionen[3]

Wichtig hierbei ist zu erkennen, dass die Annahme ein Roboter befinde sich immer im Mittelpunkt einer Zelle, zum Verschieben des eigentlichen Sicht- bzw, Kommunikations-Quadrates führt. Dadurch kann der Fall entstehen, dass Roboter nicht ermittelt werden, da sie von vornherein als *unerreichbar* gelten oder, dass zuviele Zellen (unnötige Schnittzellen) überprüft werden.
Eine Performanceerhöhung bringt diese Verwaltung lediglich erst ab einer hohen Roboterzahl und einigermaßen guten Verteilung über die gesamte Karte.

3.3. Datenstrukturen

Um die Karte und die Schätze auf der Karte mit geringer Platz- und Zeitkomplexität zu speichern, wählten wir jeweils verschiedene Implementationen der Datenstruktur *Quadtree* (vgl. [Sam90]). Diese sind in den folgenden Abschnitten vorgestellt.

3.3.1. Quadtree für Speicherung der Map

Die Anforderungen an die Datenstruktur zur Speicherung der Karte sind die folgenden:

1. Es soll ein großes Gelände im Speicher gehalten werden können, also spielt Platzeffizienz eine wichtige Rolle

2. Sie soll die Möglichkeit bieten, das Gelände aus einer gegebenen XML-Struktur zu parsen

3. Sie muß sehr viele Bereichsanfragen in der Form “Welche Hindernisse befinden sich in dem Rechteck R “innerhalb kurzer Zeit exakt beantworten können

[3] Positionen, welche außerhalb der Zellenaufteilung liegen. Bei einer Zellenbreite von 16 wäre eine illegale Position in Abbildung 6 z.B. (70,10).

Als Lösung für das Speicherproblem wählten wir die Datenstruktur *RegionQuadtree*. Diese Baumstruktur zeichnet sich dadurch aus, daß jeder Knoten genau dann vier Kindsknoten erhält, wenn sich in einem von ihnen etwas befindet; ansonsten besitzt ein Knoten keinerlei Kinder. Ein einfaches Beispiel findet sich in Abbildung 7.

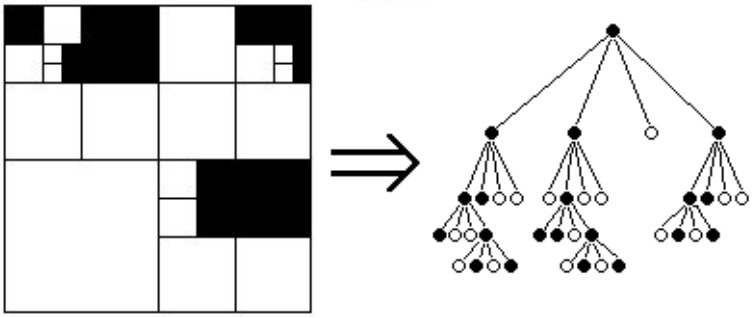


Abbildung 7: Beispiel für einen RegionQuadtree

Hier wird ersichtlich, wie eine Karte (die im konkreten Anwendungsfall natürlich um ein vielfaches komplexer aussieht) mittels Unterteilung in Viererboxen in der Datenstruktur gespeichert wird. Auf der linken Seite der Abbildung findet sich ein Kartenausschnitt; dieser wird so lange unterteilt, bis sämtliche Hindernisse (schwarz dargestellt) vollständig durch Rechtecke eingeschlossen sind. Auf der rechten Seite der Abbildung ist die Repräsentation dieser Rechtecke durch Knoten in der Datenstruktur zu sehen.
Mittels dieser Art der Speicherung wird viel Speicherplatz gespart im Gegensatz zu einer naiven Speicherung, in der beispielsweise sämtliche Hindernisse durch zwei Koordinaten in einer Liste oder ï¿½hnlichem gespeichert würden. Der Grund: Der *RegionQuadtree* bündelt die Hindernisse und speichert sie nicht punktweise. Und er speichert ausschließlich Hindernisse; leere Flächen werden nicht explizit gespeichert.
Diesen *RegionQuadtree* aus XML-Strukturen zu parsen (Anforderung 2), ist keine allzu spannende Angelegenheit: Der Parser geht rekursiv durch die XML-Struktur hindurch und instanziiert, je nach vorhandenen Hindernissen, neue Knoten oder eben nicht. Implementierungstechnisch gibt es an dieser Stelle ein wichtiges Detail: Da die XML-Struktur mittels der Open-Source-Software *JDOM* [Hun] geladen wird und dieser Vorgang viel Speicher benötigt, muß nach dem fertigen Instanziieren des *RegionQuadtrees* der Pointer auf die *JDOM*-Klassen auf **null** gesetzt werden, damit der Speicher durch den Garbage-Collector freigegeben werden kann.
Für Anforderung 3 war ein schneller Algorithmus vonnöten, der Bereichsanfragen für Rechtecke schnell beantworten konnte. Wir entwickelten dazu den folgenden Algorithmus:

1. Aus den übergebenen Anfragekoordinaten ein Anfragerechteck konstruieren

2. Für alle zu überprüfenden Knoten aus dem *RegionQuadtree* eigenes Recht-

eck konstruieren

3. Anfragerechteck und Knotenrechteck vergleichen; überschneiden sie sich => Kandidat für Hindernis im Anfragerechteck
4. Alle Kandidaten überprüfen, ob sie bereits vollständige Hindernisse sind
5. Wenn ja: Hindernisse direkt in Ergebnismenge speichern
6. Wenn nein: Die Kinderknoten von Nicht-Hindernissen rekursiv überprüfen
7. Nach der Rekursion: Ergebnismenge zurückgeben

Dieser Algorithmus arbeitet sehr effizient: Auch bei einer Simulation mit einer grosen Karte und einer hohen Roboteranzahl, also einer entsprechend grosen Datenstruktur und sehr vielen Anfragen (da die Roboter, je nach Strategie, nahezu immer mindestens 2 Bereichsanfragen pro Runde machen, erhöht sich dementsprechend die Anzahl der Anfragen) ist die Berechnungszeit dieses Algorithmus' laut Profiling-Ergebnissen in der Gesamtsimulation vernachlï¿½ssigbar.
Dieser Quadtree hat im Gegensatz zu einer normalen Quadtree-Implementierung die Eigenschaft, das jede Blatt-Zelle über ihre Nachbarzellen informiert ist. Dies ermöglicht es, Algorithmen zur Wegberechnung auf dem Quadtree laufen zu lassen.
Die Nachbarschaftsbeziehungen werden beim erstellen des Quadtrees berechnet. Dabei werden nach einer Zellteilung zuerst die vier neuen Zellen untereinander verbunden, bevor alle Nachbarn des Vaterknotens bearbeitet werden und den entsprechenden Kind-Knoten zugeordnet werden. Nach dieser Operation kennt der Eltern-Knoten keine Nachbarn mehr. Dies ist eingeführt worden, damit die Größe des Quadtree nicht überhand nimmt.

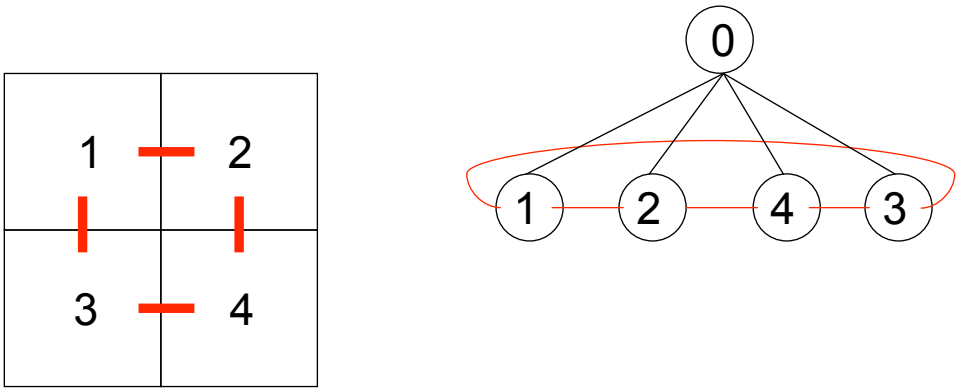


Abbildung 8: Quadtree mit Nachbarschaftsbeziehung vor Zellteilung

Im Sourcecode befindet sich die Implementation des *RegionQuadtrees* im Package `de.upb.smartteam.kernel`, und zwar in den Klassen:

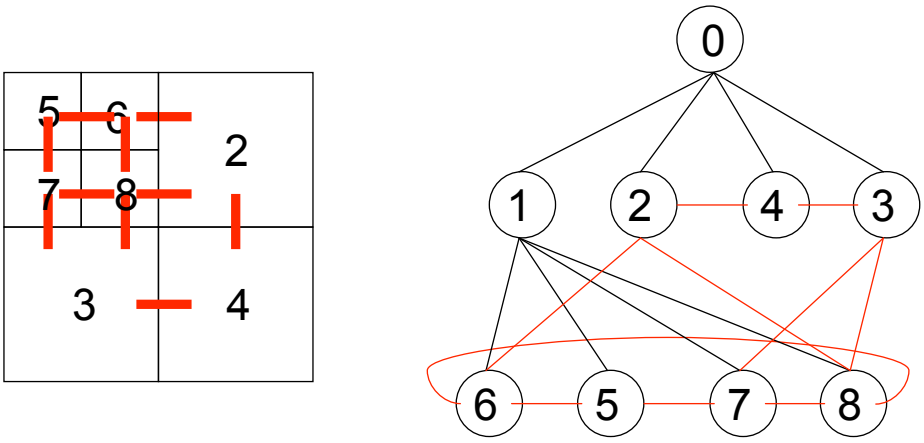


Abbildung 9: Quadtree mit Nachbarschaftsbeziehung nach Zellteilung

- `QNode`; diese enthält Kinderknoten NW, NE, SW, SE (alle vom Typ QNode), Position im Gelände, Informationen über Seitenlänge, Hinderniseigenschaft, Koordinate
- `RegionQuadtree`; diese enthält einen Wurzelknoten vom Typ QNode und Methoden zum Parsen einer XML-Geländedatei in die Datenstruktur
- `StaticObjects`; diese Unterklasse von *RegionQuadtree* erweitert diesen um den Bereichsanfrage-Algorithmus

3.3.2. PRQuadtree für Speicherung der Schätze

Es befinden sich auf der Karte nicht nur Hindernisse sondern auch andere Objekte, wie z.B. Schätze oder Flags. Die Hindernisse sind statisch, d.h. sie werden zur Laufzeit nicht mehr verändert. Die Schätze und Flags dagegen sind dynamisch, d.h. Schätze können von den Robotern abgebaut werden, Flags können an jeder Position abgesetzt werden. Daher ist es sinnvoll die Schätze und Flags getrennt von den Hindernissen zu bearbeiten. Schätze und Flags werden zu dynamischen Objekten zusammengefasst. Diese werden in einem PRQuadtree gespeichert. Eine Liste wäre in diesem Fall als Datenstruktur nicht sinnvoll, weil sie im schlimmsten Fall komplett durchlaufen werden müsste. Beim PRQuadtree wird nur ein Pfad von der Wurzel bis zu einem Knoten durchlaufen.

Aufbau eines PRQuadtrees

Der PRQuadtree unterteilt die Karte rekursiv in vier gleich große Teilbereiche, siehe Abbildung 10.
Die Informationen der gesamten Karte werden in dem Wurzelknoten gespeichert. Befinden sich mehr als ein dynamisches Objekt in einem der Knoten, so wird der Knoten bzw. der Teilbereich den der Knoten darstellt in vier kleinere

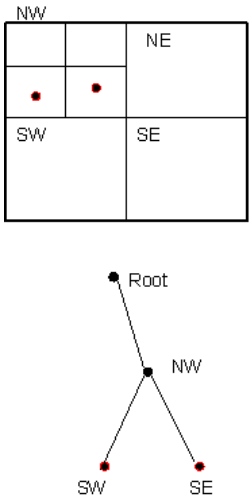


Abbildung 10: Aufbau eines PRQuadtrees

Teilbereiche unterteilt. Die Teilbereiche werden mit NW, NE, SW und SE bezeichnet, damit sie unterscheidbar sind. Nach der Teilung, werden die Bereiche als neu Kindknoten angehängt, die mindestens ein dynamisches Objekt enthalten. Enthält der neue Knoten mehr als ein Objekt, wird die ganze Prozedur wiederholt, es sei denn, die maximale Tiefe ist erreicht. Alle Blattknoten des PRQuadtrees enthalten mindestens ein dynamisches Objekt.

Klassenübersicht

Folgende Klassen wurden erstellt um die Datenstruktur aufzubauen (siehe Abbildung 11).

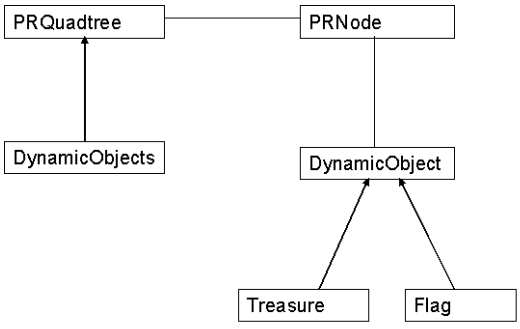


Abbildung 11: Klassenübersicht

Klasse PRQuadtree: Erstellt den Quadtree. In diesen können Objekte einegefügt oder gelöscht werden und es kann nach einem bestimmten Knoten gesucht werden.
Klasse DynamicObjects: Erbt die Funktionen von der Klasse PRQuadtree und stellt weitere zur Verfügung, wie z.B. das erstellen eines Quadtrees aus einer

*.xml-Datei. Stellt eine weitere Funktion bereit mit der man alle Objekte im Sichtradius eines Roboters bekommt und eine Funktion, die die Visualisierung über geänderte Objkete informiert.
Klasse PRNode: Erzeugt die Knoten des PRQuadtrees, kann daher entweder ein Wurzelknoten, ein Innererknoten oder ein Blattknoten sein. Verweist zudem auf die Kindknoten (NW, NE, SW, SE), falls es kein Blattknoten ist und enthält als Blattknoten dynamische Objekte.
Klasse DynamicObject: Erstellt ein Objekt, welches im Quadtree gespeichert werden kann. Jedes Objekt enthült eine eigene ID. Ein DynamicObject ist entweder ein Treasure oder ein Flag.
Klasse Treasure: Erbt von der Klasse DynamicObject. Erzeugt ein Schatz-Objekt. Diese enthült zur ID, noch den Typ und die Menge des Rohstoffes sowie die Menge die abgebaut wird.
Klasse Flag: Erzeugt die Brotkrümmel.

3.4. RMI

Um die Daten zwischen dem Kernel und Visualisierung auszutauschen, wird ein RMI-Server (Remote Method Invocation) verwendet. Der Server ist in der Klasse KernelToVisu realisiert. Die wichtigsten Funktionen:

- **deliver(VisuMessage m)** - Visualisierung holt eine Nachricht vom Kernel
- **send(VisuMessage m)** - Kernel sendet eine Nachricht an Visualisierung

Bei dem Start des Kernel startet der Server auf dem Standard-Port (Settings.java - LOOKUPPORT). Da man mehrere Server starten kann, überprüft der Server, ob der Standard-Port besetzt ist, falls dass der Fall ist, sucht er einen freien Port, in dem er einfach nächsthöheren ausprobiert. Die RMI-Verbindungsdaten können dann in der Kernel-GUI nachgesehen werden.

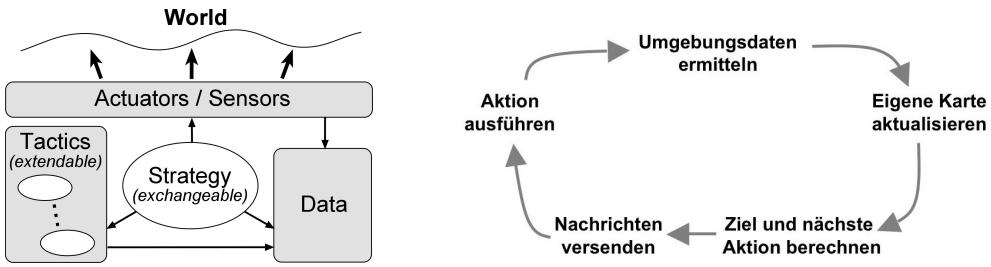


Abbildung 12: Aufbau *(links)* und Aktionszyklus des Roboters *(rechts)*

4. Roboter-Modul

In diesem Kapitel werden Architektur, wesentliche Bausteine, Strategie- und Taktikkonzept sowie die bislang existierenden Taktiken und Strategien beschrieben. Grundsätzlich basiert der Roboter auf einer unveränderbaren Komponente, die mit ihrer Umgebung mittels Aktoren und Sensoren interagiert. Dadurch werden die physikalischen Eingenschaften des Roboters simuliert. Eine austauschbare Strategie steuert das Verhalten des Roboters. Es hat Zugriff auf ein Datenmodul, indem alle relevanten Daten über die Umwelt gespeichert werden, und auf ein Taktikmodul, das eine vordefinierte, erweiterbare Menge an Taktiken zur Verfügung stellt. Abstrakt betrachtet entspricht dies dem Schaubild in Abb. 12 links. Durch diese Trennung zwischen der physikalischen Einheit des Roboters und seiner Strategie wird ermöglicht zu garantieren, dass der Roboter sich niemals inkorrekt verhält, während er seinen *Main Loop* ausführt. Dieser zyklische Prozess ist schematisch dargestellt in Abb. 12 rechts und wird im folgenden Kapitel über die Architektur detaillierter abgehandelt.

4.1. Architektur

In Abb. 13 sind die wesentlichen Klassen des Roboters dargestellt. Der Roboter besteht prinzipiell aus den vier Komponenten `RobotControl`, `PhysicalUnit`, `RobotData` und `Strategy`.
Über die `RobotControl` hat der Kernel verwaltenden Zugriff auf den Roboter. Sie initialisiert den Roboter, ist für das Typkonzept zuständig, regelt die Abläufe im Roboter und speichert die Eigenschaften des Roboters, die sie über `IRobotFeatures` zur Verfügung stellt. Weitere Information in Kapitel 4.1.1. Die `PhysicalUnit` regelt die Datenermittlung, führt Elementaraktionen aus und stellt sicher, dass sich der Roboter korrekt verhält, genauer beschrieben in Kapitel 4.1.2. Hier finden sich die die einzigen und zwar physikalischen Schnittstellen in Richtung Kernel. In der `RobotData` findet die Speicherung aller Umgebungsdaten statt, aufgeteilt in die Bereiche Nachrichten, Containerinhalt, Parameter, statische und dynamische Daten (siehe Kapitel 4.1.3). Die abstrakte Klasse `Strategy` hat über diverse Interfaces Zugriff auf die Datenhaltung. Diese Interfaces sind Inhalt von Kapitel 4.2.3. Jede konkrete Strategie erbt von `Strategy`. Die Strategie steuert das Verhalten des Roboters, sie berechnet Aktionen für die `ActionList`, kann dabei Hilfsmethoden des `BasicMotionComputer` (siehe Kapitel 4.2.3) nutzen. Außerdem kann sie auf eine beliebige Auswahl von Taktiken

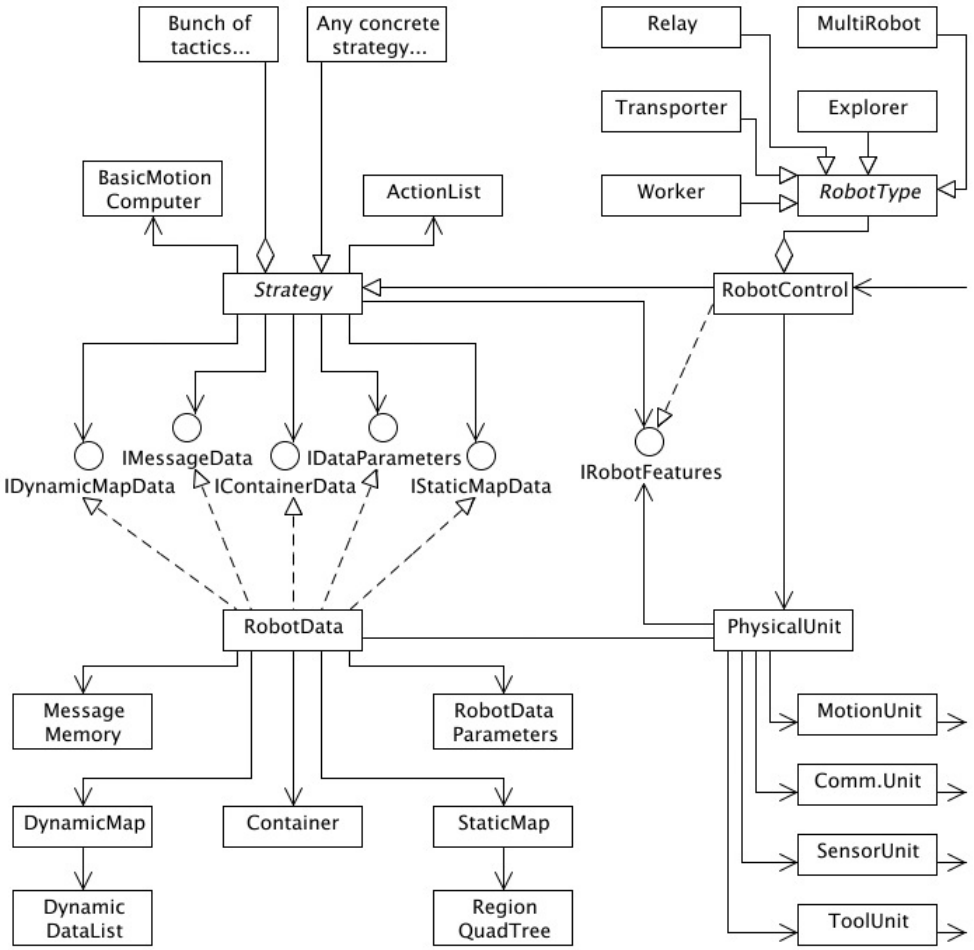


Abbildung 13: Wesentliche Klassen des Roboter-Moduls.

zurückgreifen. Die zum Taktikkonzept gehörenden Klassen sind hier ausgeblendet und werden im Kapitel 4.2.2 genauer erläutert. Das Konzept der Einbindung konkreter Strategien findet sich hingegen in Kapitel 4.2.1.
Jede Runde nun ruft der Kernel die Evaluations- und die Aktionsphase über die `RobotControl` auf. In der Evaluationsphase wird die `PhysicalUnit` dazu beauftragt, neue Umgebungsdaten ermitteln und sie in der `RobotData` zu speichern. Die Aktionsphase beginnt damit, dass die `RobotControl` die Strategie dazu auffordert, eine Elementaraktion zu berechnen. Die Strategie greift auf die benötigten Daten zu, berechnet ggf. neue Ziele und ermittelt die daraus resultierende, nächste Elementaraktion. Nebenbei speichert sie zu sendende Nachrichten im `MessageMemory`. Die `RobotControl` gibt die auszuführende Aktion weiter an die `PhysicalUnit`. Von dort aus werden die zu sendenden Nachrichten über die `CommunicationUnit` verschickt. Anschließend wird dem Kernel über die entsprechende Schnittstelle das Ausführen der Elementaraktion mitgeteilt.

4.1.1. RobotControl

Die *RobotControl* ist sozusagen das Zentralnervensystem des Roboters. Es ist der Zugriffspunkt von außen auf den Roboter (also vom Kernel). Hier werden die statischen Eigenschaften eines Roboters in Form einer `RobotConfiguration` gespeichert und können von den anderen Modulen des Roboters abgefragt werden. Weiterhin ist hier der Roboterype definiert, also z.B. Worker oder Explorer.
Die Initialisierung des Roboters läuft wie folgt ab. Eine Instanz eines Roboters kann nur durch die Klasse `RobotFactory` erzeugt werden. Diese sichert zu, dass auf den Roboter nur durch das Interface `IKernelToRobot` zugegriffen werden kann. Als erstes wird nun anhand der im Konstruktor übergebenen `RobotID` der passende Robotertyp erstellt, also eine der von `RobotType` abgeleiteten Klassen instanziiert. Beispielsweise würde für eine `RobotIDWorker` die Klasse `Worker` instanziiert. Nachfolgend wird mit der speziellen Instanz von `RobotType` geprüft, ob die Einstellungen in der `RobotConfiguration` alle Kriterien für diesen speziellen Robotertyp erfüllen, ansonsten wird eine `RobotInitException` geschmissen. Für einen Worker wäre so ein Kriterium z.B., dass die Werkzeuggröße zum Abbauen von Schätzen positiv sein muss. Zum Schlußwerden noch die Klassen `Strategy`, `RobotData` und `PhysicalUnit` instanziiert und ihre Assoziationen untereinander hergestellt.

4.1.2. PhysicalUnit

Das Package `physical` repräsentiert die physikalische Schicht des Roboters. Neben der zentralen Kontrollklasse `PhysicalUnit`, die die Ordnungsmäßigkeit aller Operationen des Roboters gewährleisten soll, beinhaltet dies die vier physikalischen Schnittstellen zum Kernel: die `MotionUnit` für Bewegungen des Roboters, die `SensorUnit` für die Ermittlung von Robotern, statischen und dynamischen Objekten im Sichtradius, die `CommunicationUnit` zum Ermitteln von Robotern im Kommunikationsradius und zum Empfangen und Versenden von Nachrichten, sowie die `ToolUnit` für das Bearbeiten, Ablegen und Aufheben von dynamischen Objekten.
In der Initialisierung erhält die `PhysicalUnit` von der `RobotControl` Zugriffe auf die `RobotData` (über das Interface `IPhysicalToData`), eingeschränkten Zugriff auf den Kernel (Interface `IRobotToKernel`) sowie die Eigenschaften des Roboters (`IRobotFeatures`) und initialisiert den Rest des Packages. Eine Aufgabe der `PhysicalUnit` ist die *Ermittlung aller Umgebungsdaten* über die passenden Schnittstellen und die Weitergabe der Daten an die `RobotData`. Dabei werden nie ungültige Daten weitergegeben, also etwa `null`, sondern im Zweifelsfall leere Listen und dergleichen. Weiter wird hier die *Ausführung von Elementaraktionen* eingeleitet, wobei sichergestellt wird, dass alle Aktionen den Fähigkeiten des Roboters entsprechen. Schließlich ist die `PhysicalUnit` der *Speicherort der korrekten Position* des Roboters. Methoden zur Verfälschung selbiger sind zwar eingerichtet, tun bislang aber nichts.
Die `MotionUnit` ist für die *Berechnung der neuen Position aus einer Bewegung* und zur Weitergabe selbiger an den Kernel zuständig. Kollisionen mit Hindernissen und Robotern werden verhindert, wofür die gewünschte Bewe-

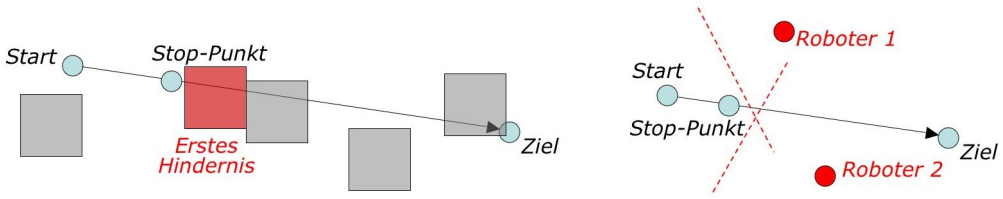


Abbildung 14: Vermeidung von Kollisionen mit Hindernissen *(links)* und Robotern *(rechts)*

gung ggf. verkürzt wird. Anschließend wird die neue Position an die `RobotData` übergeben, um den neuen Explorationszustand der Karte zu berechnen. Für die *Vermeidung von Kollisionen mit Hindernissen* wird geprüft, ob die Bewegung zu nah ($< 12,5$ cm) an einem Hindernis enden würde oder eins trifft oder kreuzt. Mittels einer Abwandlung des aus der Computergrafik bekannten *Cohen-Sutherland-Algorithmus* werden nur für relevante Fälle Schnittpunktberechnungen durchgeführt. Generell werden nur Hindernisse im Sichtradius betrachtet, was letztlich der Grund dafür ist, dass der Sichtradius größer als die maximale Geschwindigkeit des Roboters sein muss. Der Roboter bleibt vor dem nächsten Hindernis stehen (Abb. 14 links). Die *Vermeidung von Kollisionen mit Robotern* beruht darauf, dass Roboter niemals näher als $12,5$ cm an die Mittelsenkrechte zu anderen Robotern heranfahren. Dadurch können sich Roboter nicht treffen, denn alle halten sich an dieses Prinzip. Für Roboter mit Abstand maximal doppelt so groß wie die maximale Geschwindigkeit des Roboters (plus 2 mal $12,5$ cm) wird eine Schnittpunktberechnung der Bewegung mit der Mittelsenkrechte durchgeführt. Vor dem nächsten Schnittpunkt bleibt der Roboter stehen (Abb. 14 rechts). Performance-Tests haben untermauert, dass die Kollisionsvermeidung nur einen Bruchteil der Berechnungen eines Roboters darstellt (in den Tests im Bereich von 1%).

4.1.3. RobotData und Wegberechnung

Im Package RobotData (`data`) sind die Daten gekapselt, die vom Roboter benötigt werden, um Berechnungen des nächsten Schrittes durchzuführen. Während der Evaluierungsphase greift die PhysicalUnit (Package `physical`) schreibend auf die RobotData zu und aktualisiert die statischen und dynamischen Objekte, sowie die eingegangenen Nachrichten. Während der Aktionsphase greift die Strategie (Package `strategy`) dann auf die RobotData zu und holt sich notwendige Daten, um den neuen Schritt zu berechnen.
Folgende Daten gehören in die RobotData: die `StaticMap`, in der alle statischen Objekte gespeichert werden (in unserem Fall beschränkt sich das nur auf Hindernisse), die `DynamicMap`, in der dynamische Objekte gespeichert werden (Roboter und Flags), der `Container`, in dem Schätze und andere Bodenobjekte abgelegt werden können und die `MessageMemory`, die eingegangene, gesendete und zu sendende Nachrichten speichert.
Es werden weiterhin 2 Interfaces benutzt. Zum einen ist das das Interface `IRobotFeatures`, um an roboterspezifische Daten, wie zum Beispiel die ID

oder den Energieverbrauch, zu kommen. Zum anderen ist das das Interface `IRobotDataParameters`. Darüber hat man die Möglichkeit verschiedene Flags zu setzen, zum Beispiel das Speichern der Karte ein und auszuschalten.
Es gibt außerdem noch eine Referenz auf die PhysicalUnit.

Die StaticMap
Mit Hilfe der StaticMap wird der dem Roboter bekannte Bereich der Karte gespeichert. Intern wird als Datenstruktur ein Quadtree benutzt (genauer: `RegionQuadtree`). Der Quadtree wiederum besteht aus einer Menge von Knoten (`RegionQuadtreeNode`), die entweder mit dem Wert 1 (Gebiet ist bekannt) oder 0 (Gebiet ist nicht bekannt) markiert sind. Die Menge aller Knoten innerhalb des Quadtrees, die mit einer 1 markiert sind, spiegelt somit das gesamte bekannte Gebiet eines Roboters wider. Der Roboter kann nun für ein beliebiges bekanntes Gebiet erfragen, welche Hindernisse sich dort befinden. Dazu kann der Roboter Anfragen beim Kernel durchführen, und kriegt als Ergebnis die Menge der Hindernisse als Liste übergeben, die in dem angefragten Gebiet liegen (siehe Abbildung 15). Um sicherzustellen, daß der Roboter nicht Informationen über noch nicht bekanntes Gebiet erfragt, wird die Anfrage in der PhysicalUnit auf Gültigkeit überprüft. Wenn der abzufragende Bereich dabei in noch nicht erkundetes Gebiet hineinragt, wird keine Information über die Hindernisse in diesem Bereich zurückgegeben (zum Beispiel das Hindernis in der Mitte auf Abbildung 15). Die Knoten des Quadtrees haben eine feste

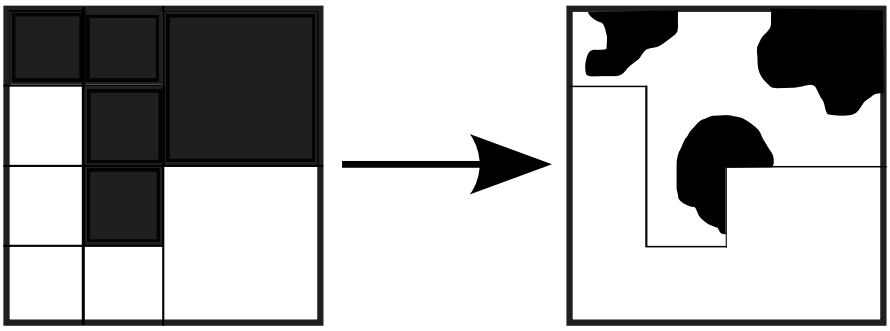


Abbildung 15: Funktionsweise der StaticMap

Mindestgröße. Dadurch kommt eine gewisse Diskretisierung der Karte zustande, und kleine Ungenauigkeiten können auftreten. Dieses ist jedoch unerläßlich, um die Funktionsweise des Quadtrees gewährleisten zu können.

Die DynamicMap
So wie in der StaticMap statische Objekte (Hindernisse) gespeichert werden, werden auch dynamische Objekte in einer eigenen Datenstruktur verwaltet. Dabei handelt es sich um die Klasse `DynamicDataList`. Die DynamicDataList ist eine Liste, in der Daten (z.B. die Roboter im Sichtbereich) gespeichert werden können, und zwar eine bestimmte Anzahl von Runden lang. Die Anzahl

der Runden, die die Daten erhalten werden sollen, ist dabei parametrisierbar. In Abbildung 16 ist ein Beispiel für eine `DynamicDataList`, in der Informationen 2 Runden lang erhalten sind.
In der DynamicMap werden nun 3 solcher Listen gehalten: eine für die Roboter im Kommunikationsradius (`robotsInCommRadius`), eine für die Roboter im Sichtradius (`robotsInViewRadius`) und eine für die dynamischen Objekte (in diesem Fall Flags) im Sichtradius (`dynamicObjectsInViewRadius`). Zusätzlich werden noch die Hindernisse im Sichtradius in einer eigenen Liste erfaßt (`staticObjectsInViewRadius`). Diese Listen werden jede Runde während der Evaluierungsphase aktualisiert.

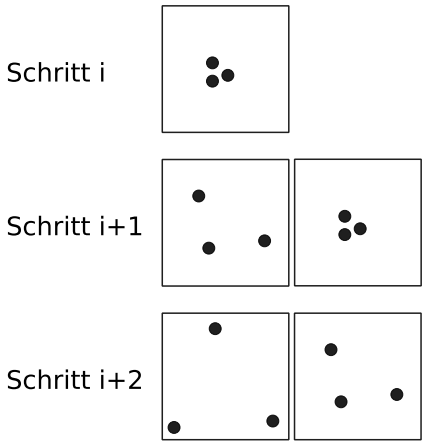


Abbildung 16: DynamicDataList

Der Container
Der Container entspricht einer Art "Ladefläche" eines Roboters. Der Roboter kann diverse dynamische Objekte in seinen Container legen, aufgenommene Objekte werden hier gelagert, und er kann gelagerte Objekte natürlich auch wieder entladen. Der Container hat eine Kapazität, das heißt, daß die Menge der Objekte, die er fassen kann, begrenzt ist.
Schätze werden dabei als Integer-Werte angesehen. Beim Ablegen eines Schatzes wird ein neues Objekt in der Landschaft mit dem gewünschten Gewicht erstellt. Danach wird die Anzahl der Schätze im Container einfach verringert. Entsprechend wird beim Aufladen die Menge der "Schätze" im Container um das Gewicht, des neuen Schatzes erhöht und das Objekt in der Landschaft entfernt. Bei den Flags ist das ein wenig anders: hier wird zwischen Inhalt und Anzahl unterschieden. Das Konzept der Anzahl entspricht dem von Schätzen, jedoch hat der Roboter ggf. zu Beginn schon eine Reihe von Flags. Außerdem kann er ein aufgehobenes Flag mit beliebigem neuen Inhalt wiederverwenden. Es wird zwischen der Menge der Schätze unterschieden, die pro Runde aufgeladen und abgeladen werden können. Diese beiden Werte können in der Roboterkonfigurationsdatei festgelegt werden.

Die MessageMemory
Da die Roboter untereinander kommunizieren sollen, ist es notwendig einen Speicher für ausgehende und auch eintreffende Nachrichten zu schaffen, so daß sie im späteren Verlauf von der Strategie analysiert werden können. Diese Aufgabe übernimmt die MessageMemory. Es gibt eine Liste, in der empfangene Nachrichten gespeichert werden (`myReceivedMessages`), eine in der bereits gesendete Nachrichten gespeichert werden (`mySentMessages`), und eine, in der noch zu sendende Nachrichten gespeichert werden (`myMessagesToSend`). Diese funktioniert wie ein Puffer - es werden im Laufe der Aktionsphase Nachrichten hinzugefügt, die dann mit Ablauf der Runde alle gesendet werden.

Wegberechnung
Um die Wegberechnungen für alle Roboter effizient durchführbar zu machen, wird diese nicht auf der Originalkarte durchgeführt, sondern auf Regionen, die miteinander verbunden sind. Der Einfachheit halber wird der schon im Kernel vorhandene Quadtree genutzt und die durch den Quadtree definierten Zellen als Regionen genutzt. Für diese Zwecke werden Kernelseitig für jede Quadtree-Zelle Informationen über deren Nachbarn berechnet. (siehe Kapitel 3.3).
Dies hat den Vorteil, das diese Aufteilung nur ein einziges mal generiert werden muss und auch nur einmal im Speicher gehalten werden muss anstatt lokal bei jedem einzelnen Roboter.
Da diese Karte nur ein einziges Mal existiert, basiert sie auf dem kompletten Gelände. Einzelne Roboter kennen nur einen eingeschränkten Teil dieses Geländes, deshalb müssen vor der Benutzung diese globalen Daten transformiert werden, um dem Wissen des Roboters zu entsprechen.
Eine komplette Transformation der Karte ist sehr zeitaufwändig ist, und hat wegen der Bewegungen der Roboter nur einen kurzen Gültigkeitszeitraum. Deshalb wird anstelle eines Preprocessing vor der Wegberechnung die Karte on-the-fly verändert. Wie sieht diese Veränderung nun aus? Grundlage für dieses Verfahren ist der Dijkstra-Algorithmus. Bevor ein Knoten im Algorithmus abgearbeitet wird, wird überprüft, ob dieser Knoten im bekannten Gebiet des Roboters liegt. Hierbei gibt es nun drei verschiedene Möglichkeiten. Die beiden trivialen Varianten, das der Knoten komplett, bzw garnicht im bekannten Gebiet liegen, betrachten wir nicht weiter.
Was passiert nun, falls ein Knoten nur teilweise im bekannten Gebiet liegt? In diesem Fall wird der Knoten in kleinere Knoten aufgeteilt (basierend auf den Zellgrenzen in der StaticMap). Diese Überdeckungen werden lokal während der Wegberechnung gespeichert und am Ende wieder freigegeben, da diese Knoten sich im Grenzgebiet befinden und sich damit wahrscheinlich bald durch die Explorationstätigkeit verändern.
Nach diesem Algorithmus erhalten wir einen Weg, dessen einzelne Knoten den Zellen oder Teilzellen des Quadtree entsprechen. Dieser muss noch in richtige Wegpunkte transformiert werden, damit die Roboter anhand derer navigieren können. Diese Wegpunkte werden jeweils in die Mitte der Strecke gelegt, die sich zwei benachbarte Zellen als Grenze teilen. Innerhalb der Startzelle und der Zielzelle werden gerade Wege vom Startpunkt bzw Zielpunkt zu dem nächsten

Wegpunkt gelegt.

4.2. Strategie- und Taktikkonzept

Eine Strategie bestimmt das Verhalten eines Roboters. Sie wird in jeder Aktionsphase des Roboters aufgerufen, um anhand neuer und bestehender Daten über die Umgebung die Ziele des Roboters zu berechnen und die dafür als nächstes auszuführende Elementaraktion zu ermitteln. Innerhalb einer Simulation kann potentiell jeder Roboter eine eigene Strategie verwenden, aber es können auch viele Roboter die gleiche Strategie haben oder zumindest eine prinzipiell gleiche und nur durch Parameter unterschiedliche gestaltete Strategie.
Im folgenden Teilkapitel (4.2.1) wird das Strategiekonzept beschrieben, das zentral für das Verständnis der Benutzung des Simulators aus Entwicklersicht ist. Anschließend wird in Kapitel 4.2.2 das Taktikkonzept erläutert, mit dem erreicht werden soll, dass immer wiederkehrende Teilstrategien wiederverwendbar zur Verfügung gestellt werden. Kapitel 4.2.3 führt dann die Schnittstellen auf, von denen ein Entwickler Gebrauch machen kann, um selbst Strategien zu erzeugen und in die Simulation einzubauen.

4.2.1. Strategiekonzept

In diesem Kapitel wird erläutert, wie das Strategiekonzept funktioniert und aufgebaut ist. Die Klasse `Strategy` spielt bei der Entwicklung von Strategien eine zentrale Rolle. Es ist eine abstrakte Klasse, die bereits Referenzen auf alle nötigen Funktionalitäten, enthält. Folgende Schnittstellen werden bereit gestellt:

- IStaticMapData (Zugriff auf die Datenhaltung für statische Karteninformationen des Roboters)
- IDynamicMapData (Zugriff auf die Datenhaltung für dynamische Karteninformationen des Roboters)
- IMessageData (Zugriff auf die Datenhaltung für Nachrichten des Roboters)
- IContainerData (Zugriff auf die Datenhaltung für Containerdaten des Roboters)
- IRobotFeatures (Zugriff auf die Eigenschaften des Roboters)
- TacticManager (Zugriff auf die verschiedenen Taktiken)
- ActionList (Zugriff auf die Aktionsliste)

Die Methoden, die diese Schnittstellen anbieten, sind in Kapitel 4.2.3 jeweils mit einer Erklärung zu Funktionalität aufgelistet. Möchte man nun eine neue Strategie implementieren, so muss diese von der Klasse `Strategy` erben. Des weiteren muss die abstrakte Methode *compute()* aus der Oberklasse (`Strategy`) in der Unterklasse (die zu implementierende Strategie-Klasse) implementiert werden.

Diese Methode setzt den wesentlichen Strategieablauf um, d.h. das gesamte Verhalten eines Roboters der die Strategie benutzt, wird in dieser Methode beschrieben. Über die oben erwähnte Schnittstellen können sämtliche Informationen angefordert werden. Die folgenden Stichpunkte sollen als kleines Beispiel dienen, wie man Informationen anfordert:

- *this.staticMapData.getCurrentPosition()*: liefert die aktuelle Position des Roboters in Koordinaten bzgl. der x -und y-Achse
- *this.staticMapData.getObstaclesInViewRadius()*: liefert Hindernisse im Sichtradius
- *this.dynamicMapData.getTreasuresInViewRadius()*: liefert Schätze im Sichtradius
- *this.messageData.getReceivedMessages()* liefert Empfangene Nachrichten
- *this.containerData.getAvailableCapacity()*: Zu Verfügung stehende Container-Kapazität
- *this.robotFeatures.getMaximumSpeed()*: Maximale Geschwindigkeit mit der sich der Roboter bewegen darf

Anhand der ermittelten Informationen können nun Entscheidungen getroffen werden. Sollte die Strategie entscheiden den Roboter zum Punkt (x1, y1) zu bewegen, so muss die geplante Aktion in die ActionList eingefügt werden. Die ActionList ist die zentrale Einheit über die der Roboter (Strtategie) mit der „Außenwelt“ agiert, d.h. jegliche Art von Aktionen werden in die ActionList eingefügt, um in den nächsten Runden von dem Kernel ausgeführt zu werden. Es stehen der Strategie folgende Aktionen zu Verfügung:

- ElementaryActionMove(speed, direction): Bewegungsaktion in der Geschwindigkeit und Richtung angegeben sind.
- ElementaryActionSendMessage(messages): Nachrichtenaktion in der die zu versendenden Nachrichten in Einer Liste angegeben werden
- ElementaryActionWorkGroundObject(groundObjectID, change): Bearbeitung eines Objektes (Schatz oder Flag)
- ElementaryActionLoadGroundObject(amount, groundObjectID): Aufladen eines Objekts in den Container
- ElementaryActionUnloadGroundObject(groundObjectID, value): Entladen von Objekten aus dem Container
- ElementaryActionDoNothing: Aktion die nichts tut. Falls die Strategie in einer Runde keine Aktion in die ActionList einfügt und die ActionList leer ist, so wird eine ElementaryActionDoNothing-Aktion automatisch in die ActionList eingefügt. Dies hat Datenstrukturspezifische Gründe.

Kommen wir zurück zu Strategie-Entwicklung. Hat man nun eine Bewegungsaktion festgelegt, so ist es nun sehr sinnvoll sicherzustellen, dass der Roboter beim Ausführen dieser Bewegung nicht mit einem Hindernis kollidiert. Für solch einen Fall stehen der Strategie eine Auswahl an Taktiken zu Verfügung. Möchte man z.B. im Falle einer Kollision die vorher festgelegte Richtung trotzdem beibehalten, ist die Taktik WalkAlongBoundaries die richtige dafür. Durch *walkAlongBoundaries.hasAction()* wird die Bewegungsaktion die an erster Stelle in der ActionList steht, auf Kollisionen mit Hindernissen untersucht. Falls die geplante Bewegung eine Kollision ergibt, liefert *walkAlongBoundaries.hasAction() true* als Ergebnis. Somit weiß man, dass die geplante Bewegung eine Kollision mit einem Hindernis darstellt. In diesem Fall kann nun mit dem Aufruf von *walkAlongBoundaries.performAction()* eine alternative Bewegung berechnet werden. Das Heißt in diesem Fall, dass die geplante Bewegungsaktion aus der ActionList entfernt wird, und eine neue Bewegungsaktion, die um das Hindernis herum führt, eingefügt wird. In der Strategie braucht man sich aber darum nicht kümmern, d.h. mit *walkAlongBoundaries.hasAction()* wird festgestellt ob eine Kollision bevorsteht und mit *walkAlongBoundaries.performAction()* wird eine Kollision vermieden, und gleichzeitig die Ziele der Strategie (bevorzugte Richtung) angestrebt. Es stehen eine Reihe von Taktiken zu Verfügung, die im Kapitel 4.3.1 nachgeschlagen werden können. Um ein vollständiges Beispiel einer Strategie einzusehen, wird die Klasse `ExplorationGreedStrategy` aus dem smarteam-Projekt empfohlen.

4.2.2. Taktikkonzept

Viele Strategien führen immer wieder die selben Teilschritte aus, wie z.B. ein Hindernis zu umrunden oder zurück zur Basisstation zu gehen. Dieses kann man auch als lokale Strategie auffassen oder kurz *Taktik*. Um dem Entwickler die Arbeit zu erleichtern wurde ein Konzept entwickelt, mit dem Taktiken gekapselt werden können. Damit brauch eine Taktik nur einmalig entworfen werden und kann dann von Entwicklern in jeder Strategie auf einfache Weise eingebunden werden. Betrachtet man diese Taktiken etwas genauer, so stellt man fest, dass sie alle in einer ähnlichen Weise funktionieren. Im Wesentlichen beschränken sich Taktiken auf zwei Typen: Bewegungen und Entscheidungen. Beispiele für Bewegungstaktiken sind das Umrunden eines Hindernisses, ein Ausweichmanöver beim Auftreffen auf einen anderen Roboter oder die Rückkehr zur Basisstation. Entscheidungen können z.B. sein, sich einer Gruppe von Robotern anzuschliessen oder aber die Erkennung einer Kollision. Diese beiden Taktik-Arten werden über verschiedene Interfaces realisiert: `IDecisionTactic` für Entscheidungen, `IMovementTactic` für automatische Bewegungen und `ITargetMovementTactic` für Bewegungen zu einem bestimmten Ziel.
Es wurde ein Framework entwickelt welches eine besonders einfache Benutzung für den Entwickler darstellt. Jede Taktik bekommt dabei zunächst einen besonders aussagekräftigen Namen der für die Art der auszuführenden Aktion steht. Der Entwickler kann dann diese Taktik über einen Verwalter laden. In jeder Runde holt sich die geladene Taktik dann automatisch alle Informationen die sie für ihre Aufführung benötigt. Das einzige was der Entwickler nur noch ma-

chen muss ist, die Taktik zu fragen ob sie aktuell ihre Aktion ausführen kann und sollte dem so sein, dann der Taktik mitteilen, dass sie die Aktion auch ausführen soll. Dies reduziert die unter Umständen sehr große Komplexität einer Taktik für den Entwickler lediglich auf den Aufruf zweier Methoden.

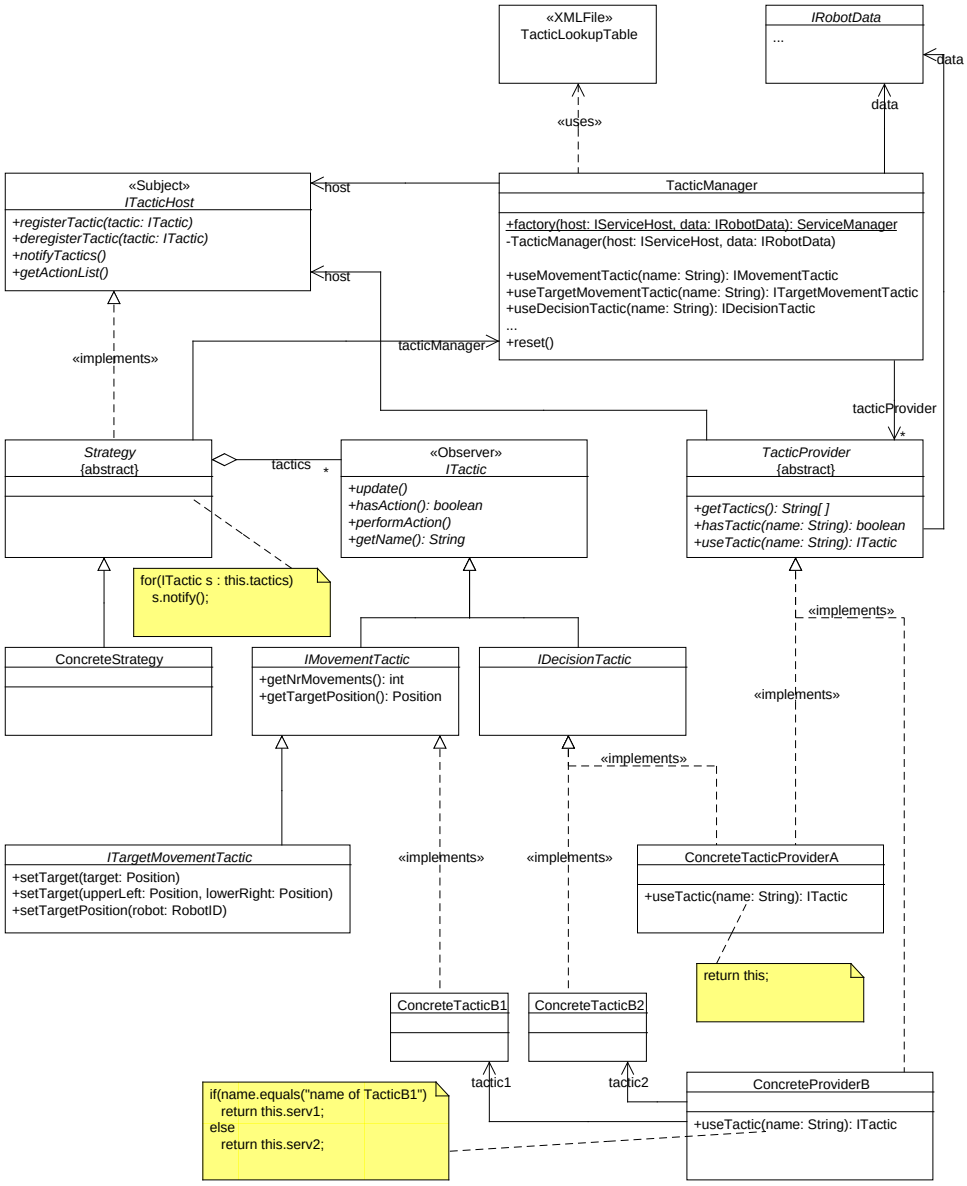


Abbildung 17: UML-Klassendiagramm des Taktik-Konzeptes

Um dies nun zu konkretisieren betrachten wir das Klassendiagramm in Abbildung 17. Der `TacticManager` ist die zentrale Verwaltung aller Taktiken. Mit ihm kann über die Methoden `useXXXTactic(name: String): IXXXTactic` eine Taktik vom Typ *XXX* angefordert werden, die den Namen `name` hat. Die Rückgabewerte sind alles Vererbungen von dem Interface `ITactic`, dem Basi-

sinterface für alle Taktik-Typen. Es beinhaltet die zwei für den Entwickler wichtigen Methoden um das Vorhandensein einer Aktion abzufragen (`hasAction(): boolean`) und um die Aktion dann auch auszuführen (`performAction()`). Die `update()`-Methode ist zur Benachrichtigung der Taktik vorgesehen, dass sich der Systemzustand geändert hat und sie neu berechnen muss, ob sie eine Aktion ausführen kann. Diese Methode wird automatisch aufgerufen von Klassen, die das Interface `ITacticHost` implementieren (wie z.B. die Klasse `Strategy`). Diese Beziehung genügt dem *Observer-Pattern* [GHJV95], wobei dies hier etwas modifiziert ist. Und zwar hat `ITactic` keine direkte Assoziation zu `ITacticHost`, sondern die abstrakte Klasse `ITacticProvider`. Dies hat folgenden Hintergrund: Taktiken können oft sehr ähnliche Funktionalität besitzen, was dazu führen könnte, dass für verschiedene Taktiken dieselben Methoden implementiert werden müssten. Um dies zu vermeiden ist jede Taktik in eine `ITacticProvider`-Klasse gekapselt. In diese Klasse stehen nun alle Methoden die die eigentliche Funktionalität enthalten und die Taktiken greifen lediglich in ihrer spezifischen Art und Weise darauf zu. Es gibt dabei zwei Möglichkeiten der Implementierung:

1. Hierbei gibt es nur eine, in ihrer Funktionalität alleinstehende Taktik, z.B. vom Typ *Entscheidung*. In diesem Fall wird eine Klasse `ConcreteTacticProviderA` vom `TacticProvider` abgeleitet, die zusätzlich das Interface `IDecisionTactic` implementiert. Damit ist der Provider auch gleichzeitig Taktik.

2. Andererseits kann es aber auch zwei sehr ähnliche Taktiken `ConcreteTacticB1` und `ConcreteTacticB2` geben, wovon eine vielleicht nur Entscheidungen trifft und die andere Bewegungen ausführt. Ihre Funktionalität kann dann in einem einzigen Provider `ConcreteTacticProviderB` gekapselt werden.

Über den Provider kann sich der `TacticManager` dann mittels der Methode `useTactic(name: String): ITactic` eine der beinhalteten Taktiken zurückgeben lassen. Damit der Manager aber überhaupt weiss welche Taktiken und zugehörige Provider es gibt, werden diese in der XML-Datei *TacticLookupTable.xml* eingetragen.

4.2.3. Schnittstellen für die Strategieentwicklung

In diesem Kapitel werden die Schnittstellen, die für den Entwurf einer Strategie relevant sind, aufgelistet und kurz erläutert welchem Zweck sie dienen.

IStaticMapData: Diese Schnittstelle ermöglicht es der Strategie auf statische Kartendaten zuzugreifen. Folgende Methoden werden bereitgestellt:

- Position getCurrentPosition(): Ermittelt die aktuelle Position des Roboters und liefert einen Rückgabewert Position in x -und y-Koordinaten.

- List<StaticObject> getKnownObstacles(Position upperLeft, Position lowerRight): Ermittelt die Hindernisse in einem Bereich, der durch eine obere Linke Position (upperLeft) und untere rechte Position (lowerRight) festgelegt wird. Sollte der Roboter den angefragten Bereich nicht vollständig

erkundet haben, so bekommt er die Hindernisse aus dem angefragten Bereich, die er tatsächlich schon mal gesichtet hat.

- <StaticObject> getObstaclesInViewRadius(): Ermittelt die aktuellen Hindernisse im Sichtradius.
- boolean isExplored(Position position): Überprüft, ob die übergebene Position bereits erkundet wurde.
- boolean isExplorationComplete(): Überprüft, ob die komplette Karte erkundet wurde.
- RegionQuadTreeNode getQuadTreeNode(Position position): Gibt den QuadTree-Knoten zurück, in dem die übergebene Position sich befindet.
- boolean isInMotionRange(Position position): Liefert *true* falls die angegebene Position in einer Runde mit maximaler Geschwindigkeit erreichbar ist.
- boolean isOnObstacleInViewRadius(Position position): Indiziert, ob die übergebene Position in einem Hindernis innerhalb des Sichtradius liegt.
- List<Position> searchPathInKnownArea(Position from, Position to): Berechnet einen kurzen (nicht unbedingt kürzesten) Weg zwischen zwei Positionen, der komplett innerhalb des vom Roboter gesehenen Gebietes liegt.
- boolean isMapBorderReached(Position curPosition, float direction, float speed): Liefert *true* falls die übergebene Position ausserhalb des Kartenrandes oder direkt auf dem Kartenrand liegt.

IDynamicMapData: Diese Schnittstelle ermöglicht es der Strategie auf dynamische Kartendaten zuzugreifen. Zu den dynamischen Kartendaten gehören Schätze, Flags (Wegmarkierungen) und Roboter. Folgende Methoden werden bereitgestellt:

- Map<TreasureID, Position> getTreasuresInViewRadius(): Ermittelt die aktuellen Schätze im Sichtradius des Roboters. Jede Schatzmine wird dabei durch eine Position und eindeutige Id in einer Map zurückgegeben.
- Map<TreasureAtHomeID, Position> getTreasuresAtHomeInViewRadius(): Ermittelt die aktuellen Schätze an der Basisstation, falls diese im Sichtradius des Roboters liegt. Hier handelt es sich um bereits eingesammelte Schätze.
- int getTreasureTransportableAmount(TreasureID treasure): Ermittelt den Anteil des spezifizierten Schatzes, der schon abgebaut ist und von einem Roboter vom Typ Transportet abtransportiert werden kann.
- int getTreasureUnworkedAmount(TreasureID treasure): Ermittelt den Anteil des spezifizierten Schatzes, der noch nicht abgebaut ist.

- int getTreasureAmount(TreasureID treasure): Ermittelt die Gesamtkapazität des spezifizierten Schatzes. Dies beinhaltet sowohl die schon abgebauten, als auch die noch unbearbeiteten Anteile des Schatzes.
- Set<RobotID> getRobotsInCommRadius(): Ermittelt die Roboter im aktuellen Kommunikationsradius.
- Map<RobotID, Position> getRobotsInViewRadius(): Ermittelt die Roboter im aktuellen Sichtradius.

Es existieren weitere Methoden in diesem Interface, die man in der Java-Dokumentation nachschlagen kann.

IMessageData: Diese Schnittstelle ermöglicht es der Strategie das Senden und Empfangen von Nachrichten. Für ein paar spezielle Nachrichten wie das Senden der eigenen Karte oder die Anfrage nach der Karte eines anderen Roboters stehen vordefinierte Methoden zur Verfügung. Folgende Methoden werden bereitgestellt:

- List<MessageInformation> getReceivedMessages(): Ermittelt die zuletzt empfangenen Nachrichten.
- int getNumberOfMapRegions(): Ermittelt die Anzahl an vom Entwickler definierten Regionen der Karte. Diese können über das Interface IDataParameters in der Methode initDataManagement jeder Strategie geändert werden.
- sendMapRegion(int horizontal, int vertical, List<RobotID> receiverIDs): Sendet den durch die Parameter definierten Bereich der Karte an die angegebenen Roboter.
- sendMapRegion(int horizontal, int vertical, RobotID receiverID): Sendet den durch die Parameter definierten Bereich der Karte an den angegebenen Roboter.
- sendMap(List<RobotID> receiverIDs): Sendet die komplette Karte des Roboters an die angegebenen Roboter.
- sendMap(RobotID receiverID): Sendet die komplette Karte des Roboters an den angegebenen Roboter.

In dieser Schnittstelle existieren weitere Methoden, die ebenfals in der Java-Dokumentation nachgeschlagen werden können.

IContainerData: Schnittstelle ermöglicht der Strategie auf die Daten des Containers zuzugreifen. Folgende Methoden stehen zu Verfügung:

- int getAvailableCapacity(): Übermittelt die aktuell verfügbare Kapazität des Containers.
- int getAmountOfTreasureObjects(): Ermittelt die aktuelle Anzahl an Schatzeinheiten, die im Container des Roboters enthalten sind.

- int getAmountOfFlagObjects(): Ermittelt die aktuelle Anzahl an Flageinheiten des Containers des Roboters.
- Map<FlagID, Object> getAllFlagContents(): Gibt eine Kopie der Inhalte aller aufgehobenen Flags zurück.
- Object getFlagContent(FlagID flagID): Ermittelt den Inhalt des Flags mit der übergebenen ID sofern vorhanden.
- removeFlagContent(FlagID flagID): Löscht den Inhalt des Flags mit der übergebenen ID sofern vorhanden.
- removeAllFlagContents(): Löscht den Inhalt aller aufgehobener Flags.

IRobotFeatures: Über diese Schnittstelle können alle wesentlichen Eigenschaften des Roboters, ebenso wie Informationen über die gesamte Simulationssituation abgefragt werden. In der folgende Auflistung werden diese vorgestellt:

- RobotID getRobotID(): Gibt die ID des Roboters zurück.
- RobotType getRobotType(): Gibt den Typ des Roboters zurück.
- float getViewRadius(): Gibt den Sichtradius des Roboters zurück.
- float getCommunicationRadius(): Gibt den Kommunikationsradius des Roboters zurück.
- float getMaximumSpeed(): Gibt die maximale Geschwindigkeit des Roboters in Metern pro Runde zurück.
- int getContainerCapacity(): Gibt die gesamte Kapazizät des Containers des Roboters zurück.
- float getEnergy(): Gibt die aktuelle Energie des Roboters zurück.
- float getEnergyConsumption(): Gibt den Energieverbrauch pro Runde des Roboters in Prozent zurück.
- float getEnergyLoad(): Gibt den Wert zurück, den ein Roboter an Energie pro Runde maximal an der Homebase aufladen kann.
- int getToolSize(): Gibt die Größe des Werkzeugs des Roboters zurück. Das Werkzeug dient dem Roboter zum Bearbeiten von Objekten, vorwiegend also zum Abbauen von Schätzen. Die Größe des Werkzeugs entspricht der maximal pro Runde bearbeitbaren Anzahl an Objekteinheiten.
- int getBucketPutSize(): Gibt die Größe der Ablegeschaufel zurück.
- int getBucketGrabSize(): Gibt die Größe der Aufnehmschaufel zurück.
- Position getStartPosition(): Gibt die Startposition des Roboters zurück, an der er sich zu Beginn der Simulation befand.

- float getMapBorderLength(): Gibt die Höhe und Breite der Karte in Metern zurück.
- int getNumberOfRobots(): Gibt die Anzahl der Roboter an, die insgesamt in der aktuellen Simulation vorhanden sind.
- int getNumberOfRobotTypes(): Gibt die Anzahl der Roboter in der aktuellen Simulation vorhanden sind zurück, die vom gleichen Typs wie der anfragende Roboter sind.
- int getRound(): Gibt die Nummer der aktuellen Runde der Simulation an.
- Random getRandom(): Gibt einen Zufallsgenerator zurück, dessen Seed für die Simulation setzbar ist.

IDataParameters: Schnittstelle, über die eine Strategie Parameter der Datenhaltung setzen kann, vor allem, um nicht benötigte Daten nicht speichern oder gar ermitteln zu lassen. Dies dient der Verringerung des Speicher- und Laufzeitbedarfs. Eine Runde in der Simulation besteht aus einer Evaluierungsphase und eine Aktionsphase. In der Evaluierungsphase werden die Datenhaltungsklassen aller Roboter aktualisiert, damit die Strategie-Berechnung bei Bedarf auf diese Daten zugreifen kann. Falls man aber zum Beispiel eine Strategie entwickelt, in der der Kommunikationsradius nie benutzt wird, so macht es Sinn diese Daten nicht in den Datenhaltungsklassen zu speichern.

- useExplorationQuadTree(boolean useQuadTree): Setzt fest, ob der Quadtree, der die erkundete Karte eines Roboters repräsentiert, verwendet werden soll. Default-Wert ist *true*.
- setBackUpsOfRobotsInViewRadius(int backUpsOfRobotsInViewRadius): Setzt fest, wieviele Runden früher im Sichtradius befindliche Roboter gespeichert werden sollen. Default-Wert ist 0.
- setBackUpsOfRobotsInCommRadius(int backUpsOfRobotsInCommRadius): Setzt fest, wie viele Runden, früher im Funkradius befindliche Roboter gespeichert werden sollen. Default-Wert ist 0.
- setBackUpsOfDynamicObjectsInViewRadius(int backUpsOfDynamicObjectsInViewRadius): Setzt fest, wieviele Runden, früher im Sichtradius befindliche dynamische Objekte gespeichert werden sollen. Default-Wert ist 0.
- saveRobotsInCommRadius(boolean saveRobots): Setzt fest, ob Roboter im Funkradius ermittelt und temporär gespeichert werden. Default Wert ist *true*.
- saveDynamicObjects(boolean saveDynamicObjects): Setzt fest, ob dynamische Objekte im Sichtradius ermittelt und temporär gespeichert werden. Default Wert ist true.

- saveMessages(boolean saveMessages): Setzt fest, ob Nachrichten empfangen und gespeichert werden sollen. Default-Wert ist *true*.
- setNumberOfMapRegions(int numberOfMapRegions): Setzt die Anzahl als solche definierter Regionen der Karte. Wird für das Versenden von Kartenteilen benötigt. Default-Wert ist 1.

TacticManager: Eine Instanz des Tacticmanagers ermöglicht es einer Strategie auf alle zu Verfügung stehenden Taktiken zuzugreifen, um diese zu benutzen. In einer zu implementierenden Strategie, die eine Unterklasse der Klasse (Strategy) ist, hat man bereits die Möglichkeit auf eine Referenz der Klasse `TacticManager` zuzugreifen. Dabei stehen folgende Methoden zu Vefügung:

- IMovementTactic useMovementTactic(String name): Gibt eine Instanz der zu verwendenden Taktik vom Typ "Bewegungßurueck.
- ITargetMovementTactic useTargetMovementTactic(String name): Gibt eine Instanz der zu verwendenden Taktik vom Typ "Bewegung zu Zielßurueck.
- IDecisionTactic useDecisionTactic(String name): Gibt eine Instanz der zu verwendenden Taktik vom Typ Ëntscheidungßurueck.
- ITeamTactic useTeamTactic(String name): Gibt eine Instanz der zu verwendenden Taktik vom Typ "Teamßurueck.
- void reset(): De-registriert alle Taktiken vom das Interface `ITacticHost` implementieren und entlaedt diese auch aus der internen Verwaltung.

Eine Besonderheit der Taktiken ist, dass sie zu Laufzeit geladen werden können und durch den Aufruf *reset()* wieder von der Strategie „entkoppelt“ werden können. In der Klasse `ExplorationGreedStrategy`, des smartteam-Projekts kann eingesehen werden wie Taktiken benutzt werden.

ActionList: Durch eine Instanz der Klasse `ActionList` werden alle einem Roboter zu Verfügung stehenden „Elementaraktionen“ der Simulation mitgeteilt, falls der Roboter beabsichtigt diese auszuführen. Die `ActionList` ist eine Datenstruktur, die ermöglicht Aktionen einzufügen, Aktionen entfernen, oder auch gannze Intervalle von Aktionen einfügen bzw. entfernen. Folgende Methoden werden bereitgestellt:

- int size(): Ermittelt die Anzahl an Aktionen in der Liste.
- boolean isIdle(): Indiziert, ob der Roboter nichts zu tun hat, d.h. die Liste ist leer.
- ElementaryAction getNextAction(): Gibt die nächste Elementaraktion ausschliesslich zu Informationszwecken zurueck.

- List<ElementaryAction> getActionInterval(int from, int to): Gibt einen Vektor mit den Elementaraktionen aus dem Intervall von from bis to zurueck falls moeglich. Ist ein Teil des Intervalls nicht vorhanden, wird nur der vorhandene Bereich zurueckgegeben.
- appendAction(ElementaryAction elemAction): Fuegt eine ganz konkrete Elementaraktion hinten an die Aktionsliste an.
- insertAction(ElementaryAction elemAction, int index): Fuegt eine ganz konkrete Elementaraktion an die Stelle index in der Aktionsliste ein.
- appendActionList(ActionList actionList): Hängt die übergebene ActionList an die bestehende an.
- insertActionList(ActionList actionList, int index): Fuegt die uebergebene ActionList in die bestehende zwischen index und index+1 ein.
- move(List<Position> pathNodes, float speed): Berechnet aus einer gegebenen Folge von Positionen eine Folge von Bewegungs-Elementaraktionen und fügt diese an die Aktionsliste an.
- move(List<Position> pathNodes, float speed, int index): Berechnet aus einer gegebenen Folge von Positionen eine Folge von Bewegungs-Elementaraktionen und fügt diese in die Aktionsliste an die Stelle ïndexëin.
- move(Position startPos, Position endPos, float speed): Fügt eine aus Start- und Endposition zu berechnende Folge von Bewegungs-Elementaraktionen an die Aktionsliste an.
- move(Position startPos, Position endPos, float speed, int index): Fügt eine aus Start- und Endposition zu berechnende Folge von Bewegungs-Elementaraktionen in der Aktionsliste vor ïndexëin. Gibt den Index der ersten Elementaraktion nach den eingefügten zurück.
- workOnObject(GroundObjectID objectID, int change, int numberOfRounds): Fügt eine sich aus der übergebenen Anzahl an Runden ergebende Folge von Bearbeitungs-Elementaraktionen an die Aktionsliste an.
- workOnObject(GroundObjectID objectID, int change, int numberOfRounds, int index): Fügt eine sich aus der übergebenen Anzahl am Runden ergebende Folge von Bearbeitungs-Elementaraktionen in die Aktionsliste zwischen index und index+1.
- loadObject(GroundObjectID objectID, int totalAmount, int amountPerRound): Fuegt eine aus Anzahl insgesamt und pro Runde aufzuhebender Objekte zu berechnende Folge von passenden Elementaraktionen an die Aktionsliste an.
- loadObject(GroundObjectID objectID, int totalAmount, int amountPerRound, int index): Fügt eine aus Anzahl insgesamt und pro Runde aufzuhebender Objekte zu berechnende Folge von passenden Elementaraktionen in der Aktionsliste zwischen index und index+1 ein.

- unloadObject(GroundObjectID objectID, int totalAmount, int amountPerRound): Fügt eine aus Anzahl insgesamt und pro Runde abzulegender Objekte zu berechnende Folge von passenden Elementaktionen an die Aktionsliste an.
- unloadObject(GroundObjectID objectID, int totalAmount, int amountPerRound, int index): Fügt eine aus Anzahl insgesamt und pro Runde abzulegender Objekte zu berechnende Folge von passenden Elementaraktionen in der Aktionsliste zwischen index und index+1 ein.
- unloadObject(GroundObjectID objectID, IFlagContent content): Fügt eine Entladeaktion eines Objektes mit dem übergebenen Inhalt an die Aktionsliste an. Für Schätze sollte ein int als Anzahl übergeben werden, für Flags ein beliebiger Inhalt.
- unloadObject(GroundObjectID objectID, IFlagContent content, int index): Fügt eine Entladeaktion eines Objektes mit dem übergebenen Inhalt in die Aktionsliste an die übergebene Stelle ein. Für Schätze sollte ein int als Anzahl übergeben werden, für Flags ein beliebiger Inhalt.
- sendMessage(ArrayList<MessageInformation> messages): Hängt eine Elementaraktion zum Versenden von Nachrichten an die Liste an.
- sendMessage(ArrayList<MessageInformation> messages, int index): Fügt eine Elementaraktion zum Versenden von Nachrichten in die Liste zwischen index und index+1 ein.
- removeFirst(): Entfernt die als nächst auszuführende Aktion aus der Liste.
- removeLast(): Entfernt die letzte Aktion aus der Liste.
- removeInterval(int from, int to): Entfernt alle Aktionen im Intervall [from,to] sofern das Intervall existiert.
- removeAll(): Entfernt alle Aktionen aus der Liste.

4.3. Taktiken

Eine ganze Reihe von vordefinierten Taktiken existiert bereits. Im folgenden finden sich diese unterteilt in die äußeren Klassen, in denen sich diese befinden. Vor allem für das Vermeiden von Kollisionen (Kapitel 4.3.1) und das Anstreben bestimmter Orte (Kapitel 4.3.2) oder Roboter (Kapitel 4.3.3) wurden Taktiken implementiert, aber auch für Team-Management (Kapitel 4.3.4) und die Energierverwaltung des Roboters (Kapitel 4.3.5).

4.3.1. CollisionDetector

Die Taktiken der Klasse `CollisionDetector` implementieren alle das Interface `IMovementTactic` und widmen sich der Vermeidung von Kollisionen mit Hindernissen und/oder Robotern und der Berechnung von Ausweichbewegungen. Die Methoden der Taktiken erlauben es im Vorhinein zu erkennen, ob

eine Bewegung in einer Kollision mit einem Hindernis oder anderen Robotern enden könnte. Die korrekte Benutzung der Taktiken gewährleistet, dass keine Kollision mit Hindernissen und/oder anderen Robotern stattfindet. Genaueres ist den enthaltenen Taktiken zu entnehmen. Alle Taktiken gehen davon aus, dass der Roboter seine als nächstes gewünschte Bewegung bereits berechnet hat und diese vorne in der `ActionList` liegt. Nur dann kann sinnvollerweise getestet werden, ob eine Kollision auftreten könnte. Dann also sollten etwaige Ausweichbewegungen berechnet werden.
Das Erkennen von Kollisionen mit Hindernissen benötigt lineare Laufzeit in Abhängigkeit zur Anzahl der Hindernisse im Sichtradius, das Erkennen von Kollisionen mit Robotern ist linear abhängig von der Anzahl der Roboter im Sichtradius. Teilweise rufen die Taktiken diese Methoden mehrfach auf.

SidestepRobotEvasion
`SidestepRobotEvasion` ist eine Taktik zum seitlichen Ausweichen anderer Roboter. Sie kann nur funktionieren, wenn alle Roboter sie verwenden. Dazu wird folgendes Verfahren bei jeder Bewegung des Roboters verfolgt:

- Prüfe, ob die gewünschte Bewegung die Mittelsenkrechte zu anderen Robotern kreuzt und finde ggf. die naheste solche Mittelsenkrechte.
- Wähle je nach Wert des Parameters “SidestepTechnique” (siehe unten) eine bestimmte Ausweichbewegung.
- Prüfe, ob die Ausweichbewegung die Mittelsenkrechte zu anderen Robotern kreuzt und führe die Bewegung nur bis zur nahesten aus.
- Es ist der ausführenden Strategie selbst überlassen, das eigentliche Ziel anzustreben.

Implizit wird auch die Kollisionsvermeidung bzgl. Hindernissen eingesetzt, um die Ausweichbewegung abzusichern. Die Vermeidung bzgl. Robotern wird offensichtlich zweimal aufgerufen. Alle anderen Operationen haben konstante Laufzeit. Sämtliche relevanten Berechnungen finden in `hasAction()` statt. Es werden praktisch keine Daten permanent gespeichert. Der Speicheraufwand ist daher zu vernachlässigen.
Der angesprochene Parameter “SidestepTechnique” erwartet einen `String`. Es stehen dabei folgende Möglichkeiten für die Ausweichbewegung zur Verfügung:

- “ParallelRight” - von der aktuellen Position aus parallel zur Mittelsenkrechte nach rechts.
- “ParallelDirected” - von der aktuellen Position aus parallel zur Mittelsenkrechte in die eigentlich angestrebte grobe Richtung.
- “AngularRight” - von der aktuellen Position aus schräg nach rechts auf die Mittelsenkrechte zu.
- “AngularDirected” - von der aktuellen Position aus schräg auf die Mittelsenkrechte zu in die eigentlich angestrebte grobe Richtung.

Alles Ausweichbewegungen stellen nur Heuristiken dar und es ist keineswegs ausgeschlossen, dass andere besser wären. Eine Erweiterung dieser Taktik wäre daher denkbar. In der Praxis hat sich bislang "AngularRight" als tauglichste Methode erwiesen, weswegen dies der Default-Wert ist. Sie ist exemplarisch in Abb. 18 dargestellt.

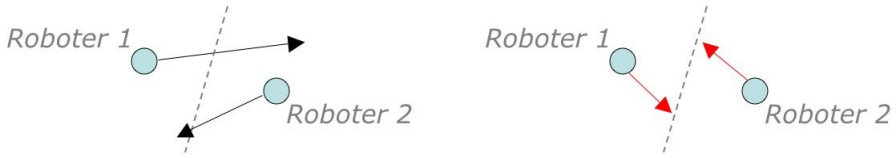


Abbildung 18: `SidestepRobotEvasion` mit Parameterwert "AngularRight" lässt Roboter schräg aneinander vorbeigehen.

WalkAlongBoundaries
`WalkAlongBoundaries` ist eine Taktik zum Umgehen von Hindernissen, basierend auf dem aus der Literatur bekannten *Pledge-Algorithmus*. Ein Hindernis wird dabei entlang dessen Wänden umgangen. Folgendes Vorgehen wird dabei verfolgt, sobald eine Bewegung eines Roboters ein Hindernis treffen würde:
Der Roboter dreht sich in eine Richtung (zu Beginn rechts) und versucht in diese weiterzugehen. Ist das nicht möglich, dreht er sich in 90-Grad-Schritten solange weiter, bis ihm kein Hindernis mehr im Weg ist (pro Drehung erhöht sich ein interner Zähler um 1). Dabei benutzt er normalerweise die bzgl. Hindernissen und Roboterfähigkeiten maximale mögliche Geschwindigkeit, es sei denn, er würde dadurch an kleine Lücken in Hindernissen vorbeilaufen. In diesem Fall reduziert er die Geschwindigkeit entsprechend. Jede Runde versucht der Roboter nun, sich wieder zurückzudrehen, sucht als nach Abbiegemöglichkeiten nach links. Ansonsten geht er weiter geradeaus oder dreht sich, wenn nötig, weiter nach rechts. Jede Linksdrehung dekrementiert den Zähler wieder um 1. Nach Pledge-Algorithmus ist das Hindernis umlaufen, wenn der Zähler den Wert 0 erreicht. Eine Ausnahme tritt auf, wenn der Roboter den Kartenrand erreicht. Dann nämlich wird die Richtung des Umgehens geändert, denn sonst würde der gesamte Kartenrand abgegangen werden. Das Vorgehen ist zum besseren Verständnis exemplarisch in Abb. 19 festgehalten.
In der Methode `hasAction()` wird maximal eine Kollisionserkennung durchgeführt und zwar nur zu Beginn des Umgehens. In `performAction()` wird der Pledge-Algorithmus aufgerufen, der im schlechten Fall das mehrmaligen Aufrufen der Kollisionserkennung nötig macht. Alle anderen Operationen sind laufzeittechnisch zu vernachlässigen, insbesondere passiert in `update()` praktisch nichts. Es werden kaum Daten permanent gespeichert, der Aufwand dafür ist daher ebenfalls zu vernachlässigen.

OrbitLeftAround und OrbitRightAround
`OrbitLeftAround und OrbitRightAround` sind Taktiken die einen Roboter veranlassen ein Hindernis zu umkreisen. Im folgenden wird die Taktik OrbitRightAround beschrieben. Durch den Aufruf der Methode hasAction() wird

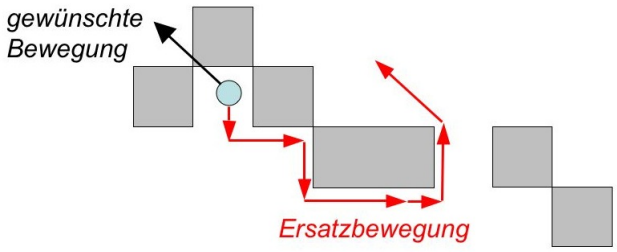


Abbildung 19: Exemplarisches Vorgehen der Taktik `WalkAlongBoundaries`

ermittelt, ob eine Kollision bevor steht oder ein Hindernis bereits umkreist wird. Falls der Rüchgabewert true liefert, kann nun durch den Aufruf der Methode performAction() einen neue Bewegungsaktion berechnet werden, die um das Hindernis herumführt. Die Berechnung wird in der Methode goRightAround() durchgeführt. Dabei wird als erstes geprüft, ob die als nächste auszuführende Bewegungsaktion in der actionList in einer Kollision mit einem Hindernis endet. Wenn das der Fall ist, wird der Winkel um 90 Grad erhöht. Das bedeutet der Roboter wendet um 90 Grad nach rechts. Der Winkel wird so lange erhöht, bis der Weg frei ist. Falls jedoch die Bewegungsaktion in der actionList kein Kollision mit Hindernissen darstellt, wird geprüft ob das wenden nach Links möglich ist. Ist das wenden möglich, wird der aktuelle Winkel um 90 Grad reduziert und die vorliegende Bewegungsaktion durch eine neue (mit dem neuen Winkel) ersetzt. Ist hingegen das Wenden nach links nicht möglich wird der aktuelle Winkel beibehalten. Der Wert der Variable obstacleReached diem ersten Aufruf von performAction() auf true gesetzt. Ist der Wert true, wird beim Aufruf der Methode hasAction() geprüft, ob eine Bewegung um 90 Grad nach links möglich ist. Wenn dies möglich ist wird geprüft ob eine Bewegung um 135 Grad nach links möglich ist. Falls dies ebenfalls möglich ist wird der Wert von obstacleReached auf false gesetzt. Das bedeutet, dass der Roboter das Hindernis nicht mehr umkreist. Diese Funktion ist wichtig, damit der Roboter nicht ewig um das Hindernis kreisen soll. Das heißt, man gibt in der Strategie eine Bewegungsaktion in die ActionList ein, die von dem umkreisenden Hindernis wegführt. Ruft man jetzt die Methode hasAction() auf, so wird diese feststellen, dass das Hindernis nicht mehr umkreist wird. Die Taktik OrbitLeftAround ist analog zu OrbitRightAround mit dem Unterschied, dass hier die Hindernisse links herum umkreist werden.

RandomObstacleEvasion
Die Taktik RandomObstacleEvasion dient dem Ausweichen von Hindernissen. In der Methode hasAction() wird geprüft ob eine Kollision bevorsteht. Wenn ja, wird die Methode randomObstacleEvasionManeuver() aufgerufen, die als Parameter die Richtung und Geschwindigkeit des Roboters enthält. Die Methode ermittelt aus 24 Richtungen die zwischen 0 und 345 Grad liegen, diejenigen, die keine Kollisionen darstellen. Aus den Gültigen Richtungen wird nun eine zufällig gewählt, die dann die die neue Richtung des Roboters darstellt. Anschließend wird eine Bewegungsaktion mit der neuen Richtung erzeugt. In der

Methode performActon() wird die aktuelle Bewegungsaktion durch die neue ersetzt, so dass der Roboter jetzt auch die neue Richtung verfolgt.

Weitere Taktiken im CollisionDetector

4.3.2. HeadForLocations

Die Taktiken in `HeadForLocations` dienen dazu, bestimmte Regionen anszustreben. Das Anstreben funktioniert grundsätzlich so, dass versucht wird, den direkten Weg zur Zielregion zu nehmen, etwaige Hindernisse im Weg des Roboter werden mit der integrierten Taktik `WalkAlongBoundaries` (siehe oben) umlaufen.

HeadForRegion
`HeadForRegion` ist eine `ITargetMovementTactic`-Taktik und dient zum Hinbewegen zu einer bestimmten, selbst zu definierenden Region. Die korrekte Benutzung der Taktik gewährleistet, dass der aktuelle Zielort erreicht wird, wenn er erreichbar ist. Es wird versucht, auf direktem Weg zur Zielregion zu gelangen. Das heißt, das nur horizontal bzw. vertikal gegangen wird, wenn sich der Roboter bereits auf Höhe der Zielregion befindet. Anderenfalls wird der aheste Eckpunkt der Zielregion angestrebt. Hindernisse im Weg des Roboters können dazu führen, dass dieses umlaufen wird.
Mittels `setTarget(Position upperLeft, Position lowerRight);` lässt sich der Taktik eine Zielregion zuweisen. Genau dann, wenn der Roboter sich nicht innerhalb der Grenzen dieser Region befindet, wird eine Aktion angeboten, um sich zu der Region hinzubewegen. Ist eine Region einmal erreicht worden, hat der Roboter sie also einmal betreten, wird sie als Zielregion gelöscht. Alle Berechnungen haben nur konstante, zu vernachlässigende Laufzeit. Ebenso werden kaum Daten permanent gespeichert.

HeadForUnexploredRegion
Die `IMovementTactic`-Taktik `HeadForCloseUnexploredRegion` dient zum Hinbewegen zu einer nahe gelegenen und (aus Sicht des Roboters) noch unexplorierten Region, die automatisch berechnet wird. Die korrekte Benutzung der Taktik gewährleistet, dass eine unexplorierte Region erreicht wird, sofern vorhanden und erreichbar.
Diese Taktik berechnet aus Gründen des Aufwands nicht die nahestgelegene Region, stellt aber eine vernünftige Heuristik dar. Zu der aktuellen Position wird der Knoten im Quadtree des Roboters ermittelt. Innerhalb des Teil-Quadtrees von dessen Elternknoten wird der größtmögliche, noch unexplorierte Knoten ermittelt. Existieren mehrere gleich große, so wird der von der aktuellen Position aus nahestgelegene solche Knoten zurückgegeben. Ist einmal eine Zielregion so berechnet worden, wird intern die Taktik `HeadForRegion` verwendet, um dorthin zu gelangen. Eine Ausnahme tritt auf wenn der Roboter neue Karteninformation von anderen Robotern empfangen hat. In diesem Fall wird erneut eine Zielregion berechnet, da sich die explorierten Teile geändert haben könnten.

Außerdem wird an der Basisstation ein neues Ziel berechnet, wenn der Parameter "NewTargetAtHomeBase" auf `true` gesetzt wird (Default-Wert `false`). Dies kann im Einklang mit der Energieveraltung stehen.
Die Laufzeit des Findens einer nahegelegenen unexplorierten Zielregion ist linear abhängig von der Tiefe des Quadtrees, also $\mathcal{O}(1)$. Sie findet bei Bedarf in `hasAction()` statt. Es wird kein relevanter Speicherplatz dafür benötigt. Alle weiteren, relevanten Berechnungen entsprechen denen von `HeadForRegion`.

4.3.3. HeadForRobots

Die Taktiken in `HeadForRobots` dienen dazu, bestimmte Roboter oder Robotertypen anzustreben. Das Anstreben funktioniert grundsätzlich so, dass versucht wird, den direkten Weg zum Zielroboter zu nehmen, etwaige Hindernisse im Weg des Roboter werden mit der integrierten Taktik `WalkAlongBoundaries` (siehe oben) umlaufen.

HeadForHomeBase Diese Taktik dient zum Hinbewegen zur HomeBase. Die korrekte Benutzung der Taktik gewährleistet, dass die HomeBase erreicht wird. Aktuell wird dazu folgende Methode verwendet: Wenn der Roboter sich nicht an der HomeBase befindet, wird eine Bewegung mit maximaler Geschwindigkeit in Richtung der HomeBase angeboten, sofern diese nicht auf ein Hindernis treffen wŕde. In diesem Fall wird die Taktik `WalkAlongBoundaries` verwendet, um das Hindernis zu umgehen. Eigentlich sollte die Wegberechnung eingebaut werden. Da sie bei Ende der Implementierungsarbeiten jedoch noch nicht stabil genug fertiggestellt war, wurde auf die Aktivierung verzeichtet. Die Vorbereitungen dafür sind im Code jedoch auskommentiert getroffen. Bei Einbau würde mittels dieser ein Weg zur HomeBase berechnet werden, wenn die Methode `hasAction()` das erste Mal aufgerufen wird. Solange dieser Weg verfolgt wird, finden keine relevanten neuen Berechnungen statt. Erst wenn von diesem abgewichen wird, muss beim nächsten Mal sinnvollerweise der Weg neu berechnet werden.
Das Vermeiden von Kollisionen mit Hindernissen hat lineare Laufzeit abhängig zur Anzahl der Hindernisse im Sichtradius. Alle anderen Berechnungen haben nur konstante, zu vernachlässigende Laufzeit. Sämtliche relevanten Berechnungen finden in `hasAction()` statt. Alle Speicherungen benötigen nur konstanten, zu vernachlässigenden Platz.

HeadForRelay
Diese Taktik dient dem Hinbewegen zu einem Relais. Die Benutzung der Taktik kann jedoch nicht gewährleisten, dass eins der Relais erreicht wird, da deren Position im Allgemeinen nicht bekannt ist. Als Ausweichmethode wird in diesem Fall die HomeBase angestrebt. Unter der Annahme, dass Relais bestimmte Positionen zur Aufrechterhaltung des Kommunikationsnetzwerks beibehalten, kann diese Taktik jedoch erfolgversprechend arbeiten.
Die Taktik liefert solange Bewegungsaktionen, bis ein Relais im Funkradius ist oder die HomeBase erreicht wurde. Ziel der Bewegungen ist dabei das

nahest vermutete Relais, ausgehend von den zuletzt gesehenen Relais. Standardmäßig werden dazu die in den letzten 1000 Runden gesehenen Relais gespeichert. Dieser Wert ist durch den Parameter "BackUpsOfSeenRelays" bei Bedarf veränderbar. Wurde an der Zielposition und auf dem Weg dorthin kein Relais gefunden, wird anstattdessen die HomeBase angestrebt, was grundsätzlich sinnvoll erscheint, da diese beim Datenaustausch hilfreich ist. Alle Berechnungen haben nur konstante, zu vernachlässigende Laufzeit. Der benötigte Speicher ist nach oben begrenzt durch die Anzahl der Relais und im schlechtesten Fall linear von diesen abhängig.

4.3.4. TeamManagement

Die Taktiken im TeamManagement geben den Robotern die Möglichkeit untereinander Teams aufzubauen/anderen Teams beizutreten, oder auch Teams wieder zu verlassen oder aufzulösen. Die Taktiken des Teammanagements arbeiten komplett auf Basis von Nachrichten[1]. Es gibt also keine zentrale Stelle, die den Aufbau der Teams organisiert, die Roboter organisieren sich vielmehr eigenständig untereinander. Entsprechend gibt es in jeder Taktik des Teammanagements auch 2 Methoden `sendQueries()` und `processQueries()`, die die Aufgabe der Nachrichtenverarbeitung übernehmen. Diese beiden Methoden werden in jeder Runde zuerst aufgerufen[2], falls es Nachrichten zu verarbeiten bzw. zu senden gibt.

Die Klassen Team und TeamConfiguration
In diesem Abschnitt sollen nochmal kurz die beiden Klassen erläutert werden, die beim TeamManagement eine große Rolle spielen: die Klassen `Team` und `TeamConfiguration`. Die Klasse `TeamConfiguration` umschreibt eine Teamkonstellation, und ist im Grunde nichts weiter als ein 5-Tupel mit Gettern und Settern. Bei der Initialisierung des TeamManagements wird die Datei *TeamConfig.xml* ausgelesen, die sich im Verzeichnis *resources* befindet. Daraus wird dann die aktuelle Teamkonfiguration erstellt. Die Konfiguration kann nachträglich jedoch mit der Methode `setConfiguration(...)` aus der `TeamBuildTactic` verändert werden.
Die Klasse `Team` modelliert ein Team. In der Klasse sind z.B. die aktuelle Konstellation des Teams und auch die gewünschte Konstellation des Teams gespeichert. Mit diesen Informationen lassen sich sehr einfach und schnell fehlende Robotertypen + deren Anzahl berechnen. In der Klasse wird außerdem der aktuelle Teamanführer gespeichert. Der Anführer hält auch zusätzlich das komplette Team, während die anderen Roboter lediglich sich selbst und den Anführer kennen. Eine Erweiterung war bislang nicht nötig, könnte jedoch verhältnismäßig einfach implementiert werden.

[1] Alle Nachrichten des TeamManagements haben einen Präfix `TM`
[2] In der Reihenfolge 1. `processQueries()` 2. `sendQueries()`

TeamBuildTactic
Mit Hilfe dieser Taktik hat der Roboter die Möglichkeit ein eigenes Team aufzubauen oder einem anderen Team beizutreten. Der Ablauf basiert auf einem Protokoll mit einem 2-fachen Handshake. Dafür sind 5 Nachrichtentypen definiert worden:

- TM_INVITE
- TM_ACK1_INVITE
- TM_NACK1_INVITE
- TM_ACK2_INVITE
- TM_NACK2_INVITE

Das Protokoll funktioniert folgerndermaßen:
Im ersten Schritt verschicken die Roboter Nachrichten vom Typ TM_INVITE. Zusätzlich wird an die Nachricht ein Gebot gekoppelt, das modellieren soll, wie "wichtig" diese Einladung für den jeweiligen Roboter sein soll. Diese Zahl ist im Moment eine Zufallszahl zwischen 0 und 1000. Kriegt ein Roboter eine Einladung, kann er darauf entweder mit der Nachricht TM_ACK1_INVITE oder TM_NACK1_INVITE reagieren. ACK1 heißt in diesem Fall, das der Roboter die Einladung annnimmt, NACK1 ist entsprechend eine Ablehnung. Kommt also ein NACK1 als Antwort, geht der Roboter nicht weiter auf diesen Fall ein. Kommt hingegen ein ACK1, gibt es wieder 2 Möglichkeiten: TM_ACK2_INVITE oder TM_NACK2_INVITE. Das ACK2 dient in diesem Fall als eine Art "Bestätigung einer Bestätigung". Da der einladende Roboter selber genauso Einladungen empfangen kann, und die Möglichkeit besteht, daß er auch eine dieser Einladungen annimmt, kann sehr wohl der Fall auftreten, daß Bestätigungen zu einer Einladung ankommen, der Roboter allerdings bereits einem anderen Roboter zugesagt hat. Für diesen Fall ist die Nachricht NACK2 gedacht.
Der Automat, der das Verhalten modelliert, ist in Abbildung 20 dargestellt.

Der Algorithmus:

```
if (messagesWaiting)
  processQueries();

if (teamNeeded OR missingTypes)
  sendQueries();
```

Im ersten Schritt schaut die Taktik nach, ob es Nachrichten gibt, die verarbeitet werden müssen. Dies ist der Fall, falls unter den eingegangenen Nachrichten mindestens eine Nachricht existiert, die den oben definierten Typen entspricht (wird durch die boolesche Variable `messagesWaiting` indiziert). Im zweiten Schritt schaut die Taktik nach, ob noch weitere Roboter benötigt werden und verschickt in dem Fall neue Einladungen. Dabei zeigt die boolesche

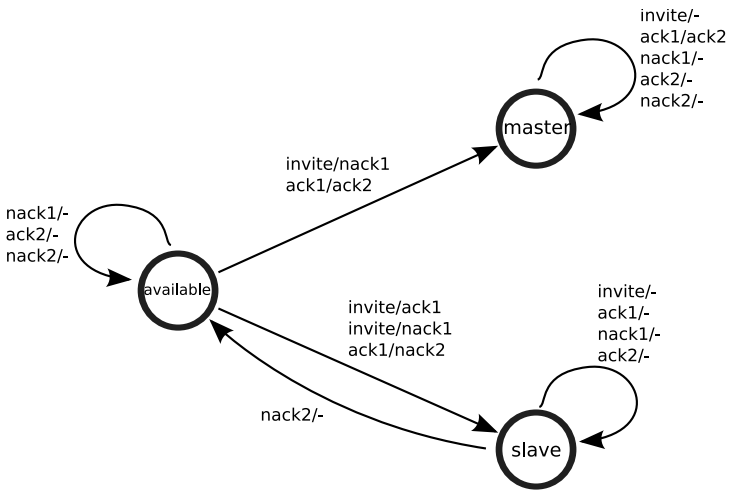


Abbildung 20: Protokoll beim Teamaufbau

Variable `teamNeeded` an, daß ein Team benötigt wird, und die boolesche Variable `missingTypes` zeigt an, ob ein Roboter, der in einem Team ist noch weitere Roboter im Team braucht (siehe `TeamConfiguration`).

`processQueries():`

1. Gehe alle Nachrichten durch, und sammle alle Einladungen (INVITE) und Bestätigungen (ACK1) von Einladungen in jeweils 2 Listen. Die Liste der Einladungen heiße `inviteList`, die Liste mit den Bestätigungen heiße `ack1List`.
2. Führe eine Auktion durch zwischen der eigenen Einladung und den fremden Einladungen durch, und bestimme so das beste Angebot.
3. Falls ich selbst die Auktion gewinne, dann verschicke an alle Roboter aus `ack1List` ein ACK2 und an alle Roboter aus `inviteList` ein NACK1. Wenn ein anderer Roboter gewinnt, dann verschicke an ihn ein ACK1, an die restlichen Roboter aus `inviteList` ein NACK1 und an alle Roboter aus `ack1List` ein NACK2.

Der Ablauf der Auktion wird momentan randomisiert durchgeführt. Um dennoch etwas auf die "Wichtigkeit" des Gebotes (das ja bei jedem INVITE mitgeschickt wird) einzugehen, wird folgendes Konzept verwendet:

Auf einem Zahlenstrahl der Länge 1 wird jedem Roboter ein Teilstück zugewiesen, das genauso lang ist, wie der Anteil seines Gebotes in Bezug zur Summe aller Gebote (auch des eigenen).
Beispiel:
Wir haben einen Roboter r_1, der Einladungen von den Robotern r_2, r_3, r_4 bekommen hat. Das Gebot eines Roboters r_i entspreche b_i. Der Roboter würfelt

danach. Der Gewinner hängt davon ab, in welchem Intervall die Zufallszahl liegt (siehe dazu Abbildung 21).

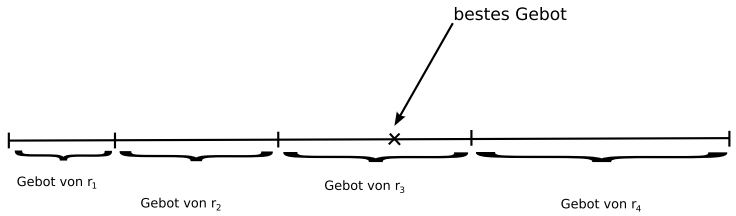


Abbildung 21: Die Auktion bei der Teambildung

`sendQueries:`

1. Suche eine Menge R passender Roboter, die sich im Kommunikationsradius befinden.
2. Generiere Gebot.
3. Versende Einladungen mit Gebot an Roboter aus der Menge R.

Auch bei den Geboten wird im Moment mit Zufallszahlen gearbeitet. Es besteht aber die Möglichkeit stattdessen eine Heuristik zu implementieren.

TeamLeaveTactic
Diese Taktik ermöglicht einem Roboter ein Team wieder zu verlassen. Diese Möglichkeit ist nur gegeben, wenn der Roboter, der austreten will, *nicht* der Anführer der Gruppe ist und natürlich auch Mitglied eines Teams ist. Für den Fall, daß es sich um den Anführer handelt, gibt es die *TeamDisbandTactic*, die etwas weiter unten erklärt wird. Die Taktik überprüft in jeder Runde zuerst, ob Nachrichten für sie eingetroffen sind, und im zweiten Schritt schaut sie nach, ob ein Austritt aus dem Team bevorsteht. Ist dies der Fall wird eine entsprechende Nachricht versendet. Die Entscheidung über einen Austritt basiert momentan noch auf einer Zufallsentscheidung.
Auch in dieser Taktik läuft der Ablauf nachrichtenbasiert, es werden folgende Typen definiert:

- TM_LEAVE
- TM_ACK_LEAVE

Das Protokoll funktioniert folgendermaßen:
Es handelt sich hierbei um ein Protokoll mit nur 1 Handshake. Ein Roboter r, der austreten will, teilt es seinem Anführer mit der Nachricht TM_LEAVE mit. Erhält dieser so eine Nachricht, bestätigt er sie grundsätzlich mit der Nachricht TM_ACK_LEAVE und nimmt den Roboter aus dem Team raus. Der Roboter

r erhält wiederum die Bestätigung, und tritt seinerseits aus dem Team aus. Der Automat, der das Verhalten modelliert, ist in Abbildung 22 dargestellt.

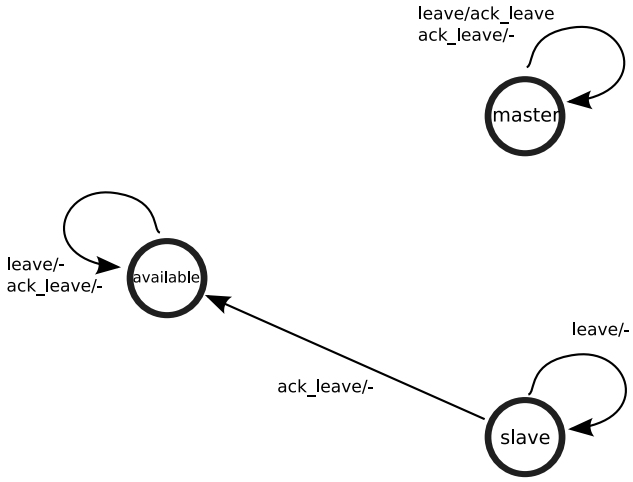


Abbildung 22: Protokoll beim Verlassen eines Teams

Der Algorithmus:

```
if (messagesWaiting)
  processQueries();

if (leaveTeam)
  sendQueries();
```

Wie auch bei der TeamBuildTactic schaut auch hier die Taktik zuerst nach, ob es Nachrichten gibt, die verarbeitet werden mössen (wird durch die boolesche Variable `messagesWaiting` indiziert). Ist dies der Fall wird die Methode `processQueries()` aufgerufen. Im zweiten Schritt wird überprüft, ob Nachrichten verschickt werden müssen. Dies ist der Fall, wenn die boolesche Variable `leaveTeam` **`true`** ist. In dem Fall wird die Methode `sendQueries()` aufgerufen.

`processQueries()`:

1. Gehe alle Nachrichten durch.

2. Kommt dabei ein Austritt (LEAVE) aus dem Team, und ich bin der Teamanführer, schicke eine Bestätigung (ACK_LEAVE) und nimm den Roboter aus dem Team. Bin ich der einzige Roboter, dann lše das Team auf.

3. Kommt eine Bestätigung über einen Austritt, den ich auch verschickt habe, dann verlasse das Team.

`sendQueries():`

1. Wenn ich nicht der Teamanführer bin, dann verschicke eine Nachricht LEAVE an den Anführer.

TeamDisbandTactic
Das Benutzen dieser Taktik ist lediglich dem Teamanführer vorbehalten, denn wie der Name schon sagt, ist sie dazu gedacht, bestehende Teams aufzulösen. Fr̈ den korrekten Ablauf wurden folgende Nachrichten definiert:

- TM_DISBAND
- TM_ACK_DISBAND

Das Protokoll funktioniert folgendermaßen:
Es handelt sich auch hierbei um ein Protokoll mit nur 1 Handshake. Ein Roboter, der auch gleichzeitig Anführer eines Teams sein muß, teilt allen seinen Teammitgliedern mit, daßdas Team aufgelöst werden soll. Dies geschieht mit der Nachricht DISBAND. Jeder Roboter, der eine solche Nachricht von seinem Teamanführer erhält, schickt an den Anführer eine Bestätigung und verlässt das Team.
Der Automat, der das Verhalten modelliert, ist in Abbildung 23 dargestellt:

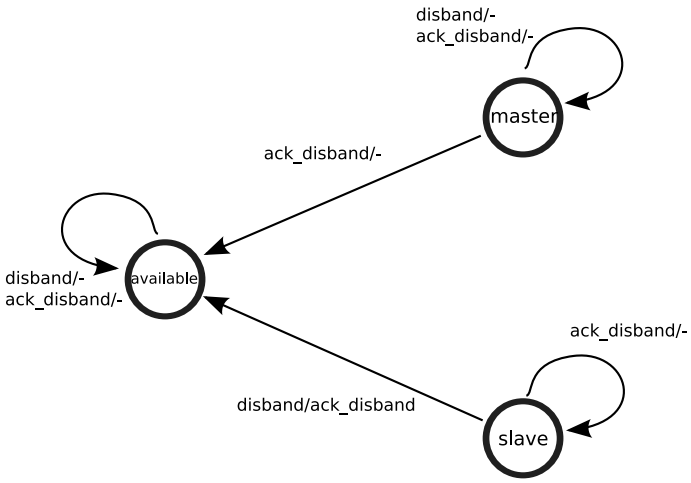


Abbildung 23: Protokoll beim Verlassen eines Teams

Der Algorithmus:

```
if (messagesWaiting)
  processQueries();

if (disbandTeam)
  sendQueries();
```

Der grundsätzliche Ablauf ist auch hier genauso, wie bei den anderen beiden Taktiken: zuerst wird auf eingegangene Nachrichten überprüft, die eventuell verarbeitet werden müssen (wird durch die Variable `messagesWaiting` indiziert), und danach werden bei Bedarf Nachrichten rausgeschickt (wird durch die Variable `disbandTeam` indiziert).

`processQueries():`

1. Gehe alle Nachrichten durch.
2. Ist die Nachricht vom Typ DISBAND, und der Sender ist mein Teamanführer, dann schicke ich eine Bestätigung und verlasse das Team
3. Wenn die Nachricht eine Bestätigung (ACK_DISBAND) ist, und ich der Anführer bin, dann nehme ich den entsprechenden Roboter aus dem Team. Wenn ich der einzige Roboter im Team bin, dann löse ich es auf.

`sendQueries():`

1. Wenn ich der Teamanführer bin, dann verschicke eine Nachricht DISBAND an alle Teammitglieder.

4.3.5. SecureEnergyManagement

Diese Taktik geht auf das Energiekonzept des Roboters ein. Die korrekte Benutzung der Taktik gewährleistet mit einer Ausnahme, dass der Roboter rechtzeitig zur HomeBase zurückkehrt, ehe seine Energie verbraucht ist. Die Ausnahme besteht in der Kollisionsvermeidung mit Robotern, da sich deren Häufigkeit nicht im Vorhinein determinieren lässt. Daher beinhalt die Taktik eine einstellbare Anzahl an Schritten, die weniger gegangen wird, als theoretisch möglich wr̈e. Die Taktik zielt darauf ab, den Berechnungsaufwand zu minimieren. Von der Verwendung der Wegberechnung wurde letztlich abgesehen, da bei Ende der Implementierungsarbeiten noch keine korrekte Benutzbarkeit der Wegberechnung in Zusammenhang mit dieser Taktik sichergestellt war. Die Vorbereitung zum Einbau sind im Code jedoch bereits vorhanden, zum Teil allerdings auskommentiert.
Mittels einer lokalen ActionList wird ein Weg zurc̈k zur HomeBase als Folge von Elementaraktionen gespeichert, der der umgekehrten Bewegung des Roboters entspricht. Dies wird in der automatisch aufgerufenen `ITactic`-Methode

`update()` sichergestellt. Sie vergleicht die aktuelle und vorherige Position und berechnet automatisch den Rückweg daraus. Auch wird hier geprüft, ob eine bestimmte Bewegung wirklich ausgeführt wurde und im negativen Fall die sich ergebende Wegänderung berechnet. An der HomeBase angekommen werden darüber hinaus solange Aufladeaktionen zur Verfügung gestellt, bis die Energie wieder aufgeladen ist. Mittels `hasAction()` kann überprüft werden, ob das Zurückgehen zur HomeBase aktuell notwendig ist bzw. ob Energie aufgeladen werden sollte. Ein einzelner Schritt des Rückwegs wird mittels `performAction()` ermittelt und automatisch in die ActionList des Roboters vorne eingefügt. Da Energieverwaltung Vorrang vor andere Tätigkeiten haben sollte, wird dazu geraten, zu Beginn jeder Runde zu prüfen, ob das Zurückgehen zur HomeBase nötig ist. Lediglich die Vermeidung von Kollisionen sollte nach Ausführen einer der von dieser Taktik angebotenen Aktionen noch sichergestellt werden.
Alle Operationen, die für die Benutzung notwendig sind, haben konstante Laufzeit. Wenn nicht die Standard-Metode `performAction()` dieser Taktik verwendet wird, kann das Löschen des Wegs an der Basisstation ggf. linerare Laufzeit in Abhängigkeit zur Länge des Wegs haben (auch abhängig von Java-Implementierung). Der Speicherplatzbedarf ist linear abhängig von der Anzahl ausgeführter Bewegungsaktionen nach dem letzten Aufladen der Energie.

4.4. Strategien

Der überwiegende Teil der Strategien ist noch in den Kinderschuhen. Da der Fokus der Projektgruppe auf der Fertigstellung des Simulators stand, konnte nur in Grenzen die Konzentration auf das Entwickeln elaborierter Strategien gerichtet werden. Die bislang erzielten Ergebnisse werden in den folgenden Teilkapiteln vorgestellt.

4.4.1. DividedExplorationStrategy

Die `DividedExplorationStrategy` stellt eine Explorer-Strategie zur Exploration der gesamten Karte dar. Die Strategie ist streng taktikbasiert und verfolgt für jeden Explorer im Groben folgendes Konzept:

- Die Karte wird imaginär in eine Anzahl an Regionen unterteilt, die der Anzahl an Explorern entspricht.
- Zu Beginn geht der Explorer zu einer durch seine ID eindeutig definierten Region.
- Diese Region soll von dem Explorer vollständig exploriert werden.
- Er geht dann zu einer von ihm aus nahe gelegenen und entsprechend seinem Wissen noch nicht vollständig explorierten Region und hilfrt dort beim Explorieren.
- Dieser Vorgang setzt sich fort bis alle Regionen exploriert sind.

Hinzukommend wird bei dieser Strategie stets versucht, die Energie des Roboters aufrechtzuerhalten. Dafür geht der Explorer regelmäßig zur HomeBase zurück, um seine Energie aufzuladen. An der HomeBase ankommend schickt der Explorer seine Karte an die HomeBase und erfragt bei selbiger den dieser bislang bekannten Gesamt-Explorationszustand. Kollisionen mit anderen Robotern und Hindernissen werden von vorneherein vermieden.
Die jede aufgerufene Methode `compute()` zeigt durch ihren geringen Umfang eindrucksvoll die Vorzüge des Taktik-Konzepts:

```
protected void compute(){
  if (energyManagement.hasAction()){
    if (!tacticToPerform.equals(energyManagement.getName()))
      this.setCurrentRegion(headForRegion);
    this.performAction(energyManagement);
  } else if (headForRegion.hasAction()) this.performAction(headForRegion);
  else {
    if (!regionExploration.hasAction() && this.isCurrentRegionNotExplored())
      this.setCurrentRegion(regionExploration);
    if (regionExploration.hasAction()) this.performAction(regionExploration);
    else if (headForUnexploredRegion.hasAction())
      this.performAction(headForUnexploredRegion);
    else if (headForHomeBase.hasAction()) this.performAction(headForHomeBase);
  }
  if (obstacleCollisionEvasion.hasAction()) this.performAction(obstacleCollisionEvasion);
  if (robotCollisionEvasion.hasAction()) this.performAction(robotCollisionEvasion);
  if (arrivesAtHome()){
    messageData.sendMap(dynamicMapData.getHomeBase());
    messageData.sendMapRequest(dynamicMapData.getHomeBase());
  }
}
```

Wichtig ist, dass diese Strategie davon ausgeht, dass die Anzahl an Explorern quadratisch ist und funktioniert nur so. Dies resultiert aus der Einteilung der Karte in quadratische Regionen. Wird eine nicht-quadratische Explorer-Anzahl gewählt, findet eine Aufteilung entsprechend der nächst größeren Quadratzahl statt. Darüber hinaus ist das Quadrat einer 2er-Potenz (1, 4, 16, 64, ...) sinnvoll, da so die einzelnen Regionen Knoten des Quadtrees entsprechen, was für die verwendeten Taktiken zum Teil Vorteile bringt.
Hinweis: Die vollständige Exploration einer Region soll eigentlich von dem Explorer mit einer aus der `SubAreaStrategy` (siehe 4.4.2) hervorgehenden Taktik umgesetzt werden. Dies konnte jedoch noch nicht mehr rechtzeitig eingebaut werden. Jedoch ist die Benutzung der Taktik `HeadForCloseUnexploredRegion` ein behelfsmäßig akzeptabler Ersatz, da auch sie natürlich zu Explorationszwecken benutzt werden kann. Ein weiteres Problem besteht in größen Hindernissen, da sie nach Umlaufen nicht als exploriert erkannt werden. Hierfür sind jedoch sinnvollererweise Maßnahmen in den festen Teilen des Roboters zu treffen.

4.4.2. SubAreaStrategy

Die *SubAreaStrategy* ist eine Strategie um einen rechteckigen Ausschnitt einer Karte (Subarea) vollständig zu explorieren. Sie ist eine systematische Suche in der Hinsicht, dass die einen vordefinierten Weg geht und Hindernisse ebenfalls nach einem vordefinierten Schema exploriert. Sie arbeitet dabei völlig unabhängig von anderen Robotern und speichert sich das bislang explorierte Gebiet in einer eigenen Datenstruktur. Nachfolgend wird die Strategie anhand von Bildern beschrieben und parallel dazu der korrespondierende Zustandsautomat in Abbildung 24 erklärt. Aus Gründen des Umfangs kann nicht auf die Bedeutung aller Transitionen eingegangen werden. Die Implementierung der Strategie wurde in Form eines *State Patterns* [GHJV95] vorgenommen.
Nehmen wir einen rechteckigen Ausschnitt einer Karte wie in Abbildung 25. Die Begrenzung der SubArea ist durch die gestrichelte Linie dargestellt, Hindernisse sind grau hinterlegt. Die blauen Streifen (*Stripes*) repräsentieren eine horizontale Unterteilung der SubArea in Abschnitte. Die Stripes sind genau im doppelten Sichtradius des Roboters entfernt. Das bedeutet, wenn der Roboter jeden Streifen abgelaufen hätte, dann würde er die SubArea vollständig explorierte haben. Die Punkte an den Enden der Streifen sind *StripeEndCheckpoints*, welche dem Roboter indizieren, dass er am Anfang bzw. Ende eines Stripes ist. Das Problem an dieser SubArea sind offenbar die Hindernisse. Es muss nicht nur das obere Hindernis irgendwie umrundet werden, sondern auch die begehbaren Bereiche unter dem unteren Hindernis exploriert werden. Dazu muss die SubArea in einer sinnvollen Weise verlassen uns wiederbegangen werden.
Angenommen der Roboter befindet sich nun zunächst außerhalb der SubArea (*OutOfAreaState*, Abb. 24) und betritt diese nun von irgendeinem Punkt aus. Die Strategie initialisiert in diesem Moment die Datenstruktur *StripedArea*, welche die Stripes erstellt und verwaltet. Dann bewegt sich der Roboter zum nächstgelegenen StripeEndCheckpoint eines der äußeren beiden Stripes (*MoveToStartStripeState*, Abb. 24). Angenommen die Strategie ist an dem rechtsoberen StripeEndCheckpoint angekommen – in Abbildung 26 entspricht dies Punkt A. Nun wird der Roboter versuchen auf direktem Weg an das entgegengesetzte Ende des Stripes zu gelangen (*ExploreStripeState*, Abb. 24). Am Punkt B bemerkt die Strategie ein unbekanntes Hindernis und setzt daher einen *ObstacleCheckpoint* um es auf diesem Stripe als gesehen zu markieren. Zusätzlich wird das bis hierhin entlanggelaufene Segment des Stripes als exploriert markiert. Dann geht die Strategie in einer festgelegten Richtung an dem Hindernis entlang (*MoveForwardAlongObstacleState*, Abb. 24) und trifft schließlich auf den Rand der SubArea, markiert durch Punkt C. Hier wird ein *BorderCheckpoint* gesetzt, der später noch gebraucht wird. Die Strategie beschließt umzudrehen und das Hindernis in der anderen Richtung entlangzugehen (*MoveBackwardAlongObstacleState*, Abb. 24). Auf ihrem Weg setzt sie bei jedem Auftreffen auf einen Stripe einen ObstacleCheckpoint, falls noch keiner existiert. Wenn sie an Punkt D angelangt ist kehrt sie zum Ausgangspunkt B zurück. Durch die zwei BorderCheckpoints ist nur klar, dass das Hindernis die Subarea teilt. Die Strategie würde nun zum Punkt E gehen (*MoveToNextObstaclePointState*, Abb. 24), den dortigen Stripe nach oben entlanggehen

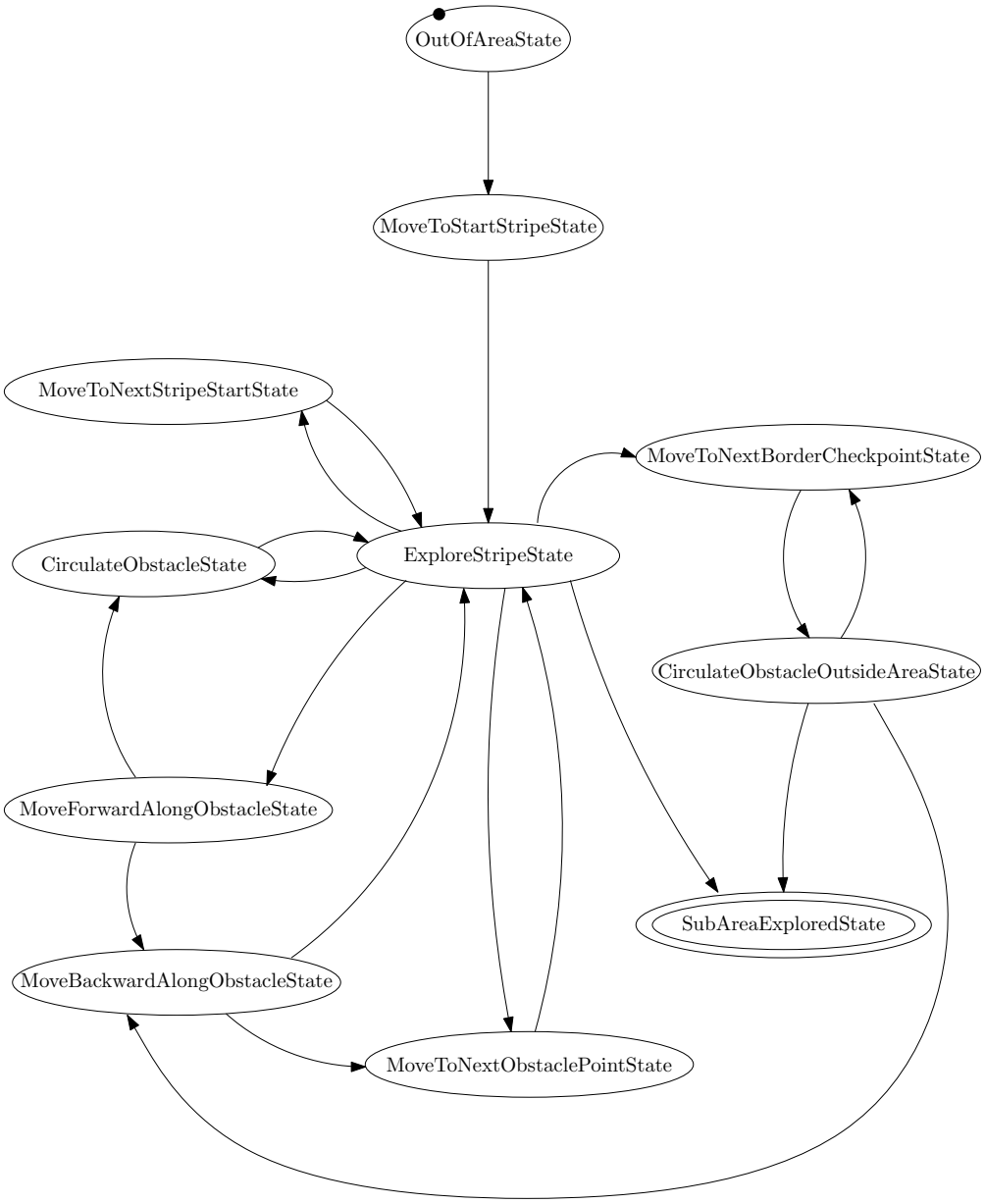


Abbildung 24: Zustandsautomat der Strategie *SubAreaStrategy*

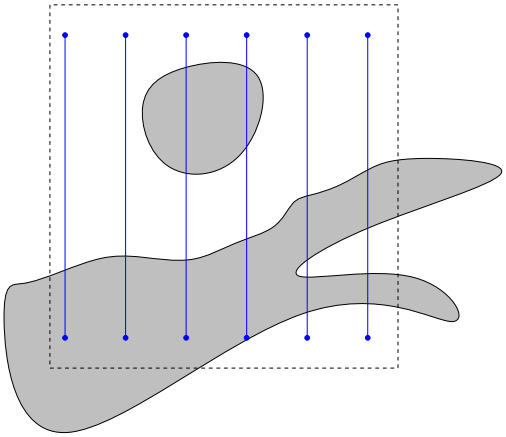


Abbildung 25: Beispiel einer SubArea mit Stripes

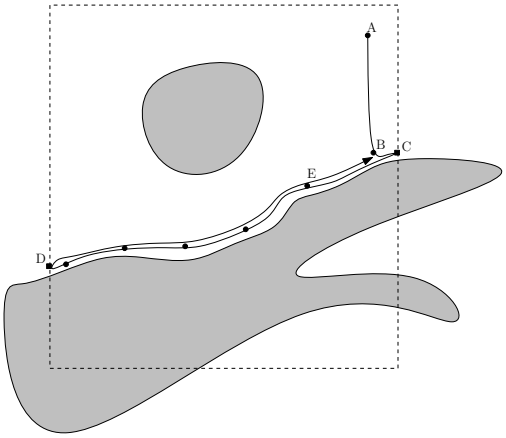


Abbildung 26: Beispiel zum Ablaufen eines Hindernisses bis zum Rand der SubArea

(*ExploreStripeState*, Abb. 24) und mit dem allgemeinen explorieren fortfahren. Wenn die Strategie am Ende des Stripes angelangt ist, geht sie einfach zu nächsten über (*MoveToNextStripeStartState*, Abb. 24).
Nun kann auch der Fall auftreten, dass beim Entlanglaufen an einem Hindernis kein Rand der SubArea getroffen wird. Dieser Fall ist in Abbildung 27 dargestellt. Der Roboter läuft von oben auf das Hindernis zu und setzt einen ObstacleCheckpoint an Punkt A. Wie zuvor läuft die Strategie das Hindernis in einer festgelegten Richtung ab und setzt ObstacleCheckpoints wann immer sie einen Stripe kreuzt. Bevor sie allerdings wie zuvor auf einen Rand der SubArea treffen kann, erreicht sie erneut den Ausgangspunkt A. An dieser Stelle existiert in jedem Fall eine gerade Anzahl von ObstacleCheckpoints, womit die innenliegenden Segmente des Stripes als exploriert markiert werden können. Nun weiss die Strategie, dass das Hindernis bekannt ist und kann es direkt umrunden (*CirculateObstacleState*, Abb. 24) um ab Punkt B den Rest des Stripes zu erkunden.
Die Frage ist nun was passiert, wenn die Strategie den letzten Stripe in der zuvor

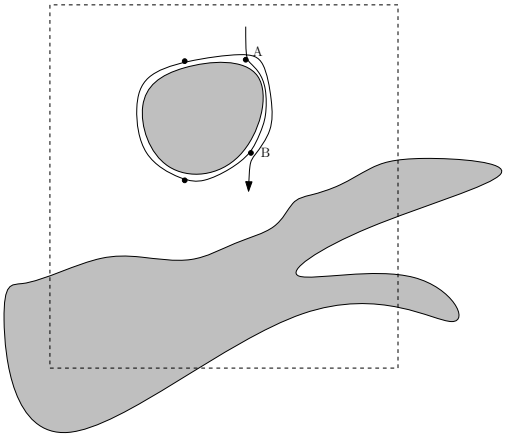


Abbildung 27: Beispiel zum Umrunden eines Hindernisses

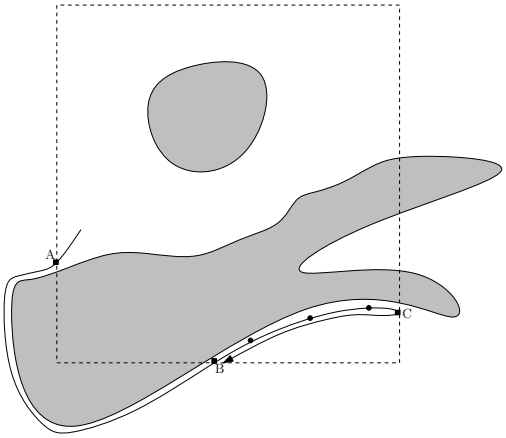


Abbildung 28: Beispiel zum Verlassen der SubArea und Erkunden von abgetrennten Bereichen

beschreibenen Weise abgelaufen hat. Hier brauchen wir nun die BorderCheckpoints. Wir gehen in diesem Fall immer zum zuletzt gesetzten BorderCheckpoint zurück (*MoveToNextBorderCheckpointState*, Abb. 24). In Abbildung 28 ist das Punkt A. Von hieraus wird das Hindernis ausserhalb der SubArea umrundet (*CirculateObstacleOutsideAreaState*, Abb. 24), bis an Punkt B wieder die SubArea betreten wird und ein neuer BorderCheckpoint gesetzt werden kann. Wie zuvor wird das Hindernis entlanggegangen (*MoveBackwardAlongObstacleState*, Abb. 24) bis zum Auftreffen auf den nächsten Randpunkt C. Nachdem zum Ausgangspunkt B zurückgekehrt wurde, kann mit dem normalen Explorieren dieses Unterabschnittes fortgesetzt werden. Dieses Verfahren wird analog für den verbleibenden Unterabschnitt angewandt. Danach ist die SubArea vollständig exploriert (*SubAreaExploredState*, Abb. 24).
Hinweis: Die Strategie kommt bislang mit zwei wesentlichen Einschränkungen, die Aufgrund des zeitlichen Umfangs nicht mehr implementiert wurden konnten. Der Fall, dass ein Hindernis nicht ausserhalb der SubArea umgangen werden kann, weil es an den Kartenrand grenzt wird nicht behandelt. Weiterhin gibt

es bislang noch Probleme mit dem korrekten setzen der ObstacleCheckpoints, wenn ein Stripe direkt (bzw. mit Toleranz des Bewegungsradius) auf dem Rand eines Hindernisses verläuft.

4.4.3. ExplorationGreedStrategy

Bei dieser Strategie ist das Ziel die Erkundung der Karte, dementsprechend eignet sich diese für Roboter vom Typ Multirobot oder Explorer. Zu Beginn wird eine Richtung zwischen 0 und 359 Grad mit einem Zufallsgenerator erzeugt. Es wird im Konstruktor der Klasse eine Bewegungsaktion (ElementaryActionMove) erstellt bei der die Richtung dem generierten Winkel und die Geschwindigkeit der maximal zulässigen Geschwindigkeit des Roboters entspricht. Diese Bewegungsaktion wird in eine Liste der Roboteraktionen (ActionList) hinzugefügt. Als erstes wird nun überprüft, ob mit der aktuellen Position, Richtung und Geschwindigkeit des Roboters der Kartenrand erreicht bzw. überschritten wird. Falls dies der Fall ist, so wird nun in einer while-schleife eine neue Richtung generiert und anschließend überprüft, ob unter Einbehaltung dieser Richtung die Bewegung auf dem Kartenrand oder in einem Hindernis endet. So bald eine gültige Richtung generiert wurde, wird die while-schleife abgebrochen. Sollte die als nächst auszuführende Bewegungsaktion nicht den Kartenrand erreichen bzw. überschreiten, wird überprüft ob die Bewegung in einem Hindernis endet. Falls dies der Fall ist, wird nun mit Hilfe der Taktik WalkAlongBoundaries der Roboter um das Hindernis herum geleitet bis die angestrebte Richtung wieder Hindernisfrei ist. Bei einer Hindernisfreien Richtung wird diese so lang beibehalten, bis einer der oben beschriebenen Fälle (Kollision mit Hindernis oder Karetnrand) eintritt.

4.4.4. TreasureGreedStartegy

Bei dieser Strategie ist das Ziel die Erkundung der Karte, Lokalisierung, Abbau und Abtransport von Schätzen. Der Roboter befindet sich stets in einem der folgenden Zustände:

- SearchTreasureState (Exploration der Karte, Suche nach Schätzen)
- WalkToTreasureState (Roboter bewegt sich zum gefundenen Schatz)
- WorkTreasureState (Schätze abbauen)
- LoadTreasureState (Schätze aufladen)
- WalkToHomeBaseState (Roboter bewegt sich zu HomeBase)
- UnloadTreasureState (Schätze abladen)

Zu Beginn der Simulation startet der Roboter im Zustand *SearchTreasureState*, d.h. er erkundet die Karte und ist auf der Suche nach Schätzen. So bald sich ein Schatz im Sichtradius des Roboters befindet, wechselt die Strategie den Zustand nach *WalkToTreasureState*. In diesem Zustand verfolgt der Roboter den direkten Weg zum gefundenen Schatz. Sollten dabei Hindernisse im Weg sein,

so werden diese mit Hilfe der Taktik WallkAlongBoundaries umgangen. So bald die Position des Schatzes erreicht ist, wird überprüft, ob Schatz-Einheiten noch vorhanden sind, denn während der Roboter sich auf dem weg zum Schatz befand, hätte ein anderer Roboter den Schatz abbauen und möglicherweise auch abtransportieren können. Falls beim Erreichen der Schatzposition kein Schatz mehr vorhanden ist, wechselt die Strategie in den Zustand *SearchTreasureState*. Falls jedoch Schatz-Einheiten noch vorhanden sind, wird überprüft ob abgebaute Schatz-Einheiten vorhanden sind, und wenn ja, wird die Menge mit der Container-Kapazität des Roboters verglichen. Reichen die bis jetzt abgebauten Schatz-Einheiten noch nicht aus um den Container zu füllen, wechselt die Strategie in den Zustand *WorkTreasureState* um weitere Schatz-Einheiten abzubauen. Falls die Abgebauten Einheiten den Container füllen oder die letzten Schätze abgebaut wurden, wechselt die Strategie in den Zustand *LoadTreasureState*. Nun ist der Roboter dabei Schätze aufzuladen. So bald der Container gefüllt ist oder die letzten Schätze aufgeladen wurden, wechselt die Strategie in den Zustand *WalkToHomeBaseState*. Der Roboter befindet sich nun auf dem Weg zu Homebase um die Schätze da zu entladen. Sollte der Roboter auf dem Weg zu Homebase auf Hindernisse stoßen, so werden diese mit Hilfe der Taktik WallkAlongBoundaries elegant umgangen. Hat der Roboter die Homebase ereicht, wechselt die Strategie nun in den Zustand *UnloadTreasureState*. In diesem Zustand bleibt die Strategie so lange, bis die Schätze aus dem Container vollständig entladen sind. So bald die Schätze entladen sind, wird durch den Aufruf *this.host.hasRemainingTreasure()* abgefragt, ob an der zu letzt besuchten Schatzposition beim verlassen noch Schätze vorhanden waren. Falls dies der Fall ist, wechselt die Strategie in den Zustand *WalkToTreasureState*, um die restlichen Schätze abzutransportieren. Falls jedoch beim verlassen der Schatzposition keine Schätze mehr vorhanden waren, wechsel die Strategie in den Zustand *SearchTreasureState* und ist somit auf der Suche nach neuen Schätzen. Der gesamte Ablauf ist der Abbildung 29 zu entnehmen.

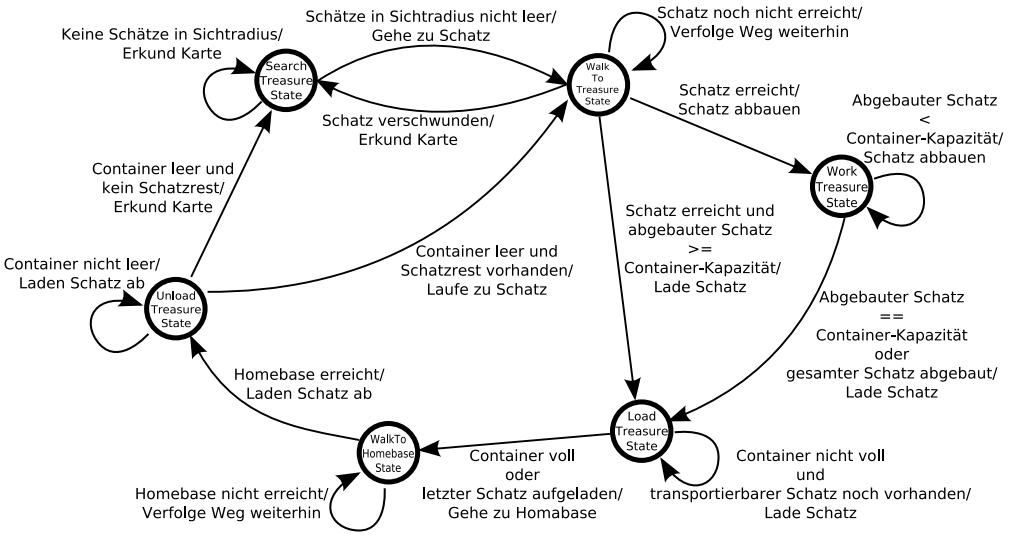


Abbildung 29: Zustandsautomat der Strategie TreasuregreedStrategy

4.4.5. WorkerStrategy

Strategie, die für Roboter vom Typ Worker oder Multirobot geeignet ist. Ein Roboter der diese Strategie benutzt, ist also in der Lage die Karte zu erkunden, und falls er dabei Schätze lokalisiert, werden diese abgearbeitet. Zweck dieser Strategie ist, für einen Roboter vom Typ Transporter die Schätze aufbereiten, so dass dieser beim erreichen der Schatzpositionen die Schätze aufladen und zu HomeBase transportieren kann. Ein Roboter der die Strategie `WorkerStrategy` benutzt befindet sich zu jedem Zeitpunkt in einem der folgenden Zustände:

- SearchTreasureState (Exploration der Karte, Suche nach Schätzen)
- WalkToTreasureState (Roboter bewegt sich zum gefundenen Schatz)
- WorkTreasureState (Schätze abbauen)

Zu Beginn der Simulation befindet sich der Roboter im Zustand *SearchTreasureState*, d.h. er erkundet die Karte und hält Ausschau nach Schätzen. So bald ein Schatz in seinem Sichtradius auftaucht, wechselt die Strategie in den Zustand *WalkToTreasureState*. Nach jeder Runde wird überprüft, ob der Roboter den Schatz erreicht hat. So bald die Position des Schatzes erreicht ist, wechselt die Strategie in den Zustand *WorkTreasureState*. Ist der Schatz vollständig abgearbeitet, wechselt die Strategie in den Zustand *SearchTreasureState* und der Roboter befindet sich erneut auf Schatzsuche. Sollte der Roboter, wähtend er sich im Zustand *SearchTreasureState* oder *WalkToTreasureState* befindet, auf Hindernisse stoßen, so werden diese mit Hilfe der Taktik WallkAlongBoundaries überwunden, ohne dabei Kollisionen zu verursachen. In Abbildung 30 ist der gesamte Ablauf noch mal als Zustandsautomat dargestellt.

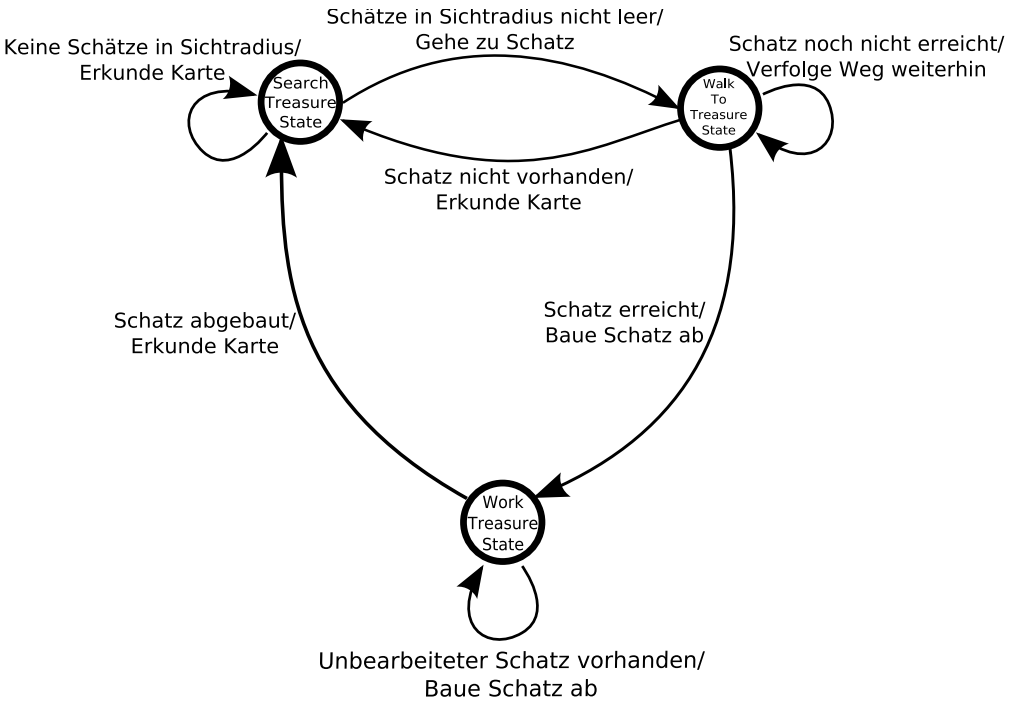


Abbildung 30: Zustandsautomat der Worker-Strategie

4.4.6. TransporterStrategy

Diese Strategie wurde für Roboter vom Typ Transporter oder Multirobot konzipiert. Ein Roboter der diese Strategie benutzt, befindet sich zu jeder Zeit in einem der folgenden Zustände:

- SearchTreasureState (Exploration der Karte, Suche nach Schätzen)
- WalkToTreasureState (Roboter bewegt sich zum gefundenen Schatz)
- LoadTreasureState (Schätze aufladen)
- WalkToHomeBaseState (Roboter bewegt sich zu HomeBase)
- UnloadTreasureState (Schätze abladen)

Zu Beginn der Simulation befindet sich der Roboter im Zustand *SearchTreasureState*. Dabei wird die Karte erkundet mit dem Ziel Schätze, die bereits von einem Roboter vom Typ Worker oder Multirobot abgearbeitet wurden und zum Abtransport bereit liegen, zu finden. Falls der Roboter bei der Erkundung der Karte auf solchen Schatz trifft, d.h. in seinem Sichtradius befindet sich ein Schatz, wechselt die Strategie in den Zustand *WalkToTreasureState*. So bald die Position des Schatzes erreicht ist, wechselt die Strategie in den Zustand *LoadTreasureState*. In diesem Zustand werden nun pro Runde eine bestimmte Menge an Schatzeinheten in den Container des Roboters geladen. In jeder neuen Runde wird überprüft, ob der Container bereits voll geladen ist. So bald dies der Fall ist, wechselt die Strategie in den Zustand *WalkToHomeBaseState*. In

diesem Zustand befindet sich der Roboter auf direktem Wege zu der Homebase. In jeder Runde wird geprüft, ob die Position der Homebase bereits erreicht ist. Falls auf dem Weg Hindernisse auftauchen, werden diese mit Hilfe der Taktik WallkAlongBoundaries umgangen, und es wird weiterhin die Position der Homebase angestrebt. So bald die Homebase erreicht ist, wechselt die Strategie in den Zustand *UnloadTreasureState*. Nun werden in jeder Runde eine bestimmte Menge an Schatzeinheiten abgeladen. Die Menge, die pro Runde abgeladen werden kann, ist durch die Größe der Abladeschaufel definiert. Diese Größe kann in der xml-Datei zu den Robotereigenschaften unter BucketPutSize eingestellt werden. So bald der Container leer ist, wechselt die die Strategie in den Zustand *WalkToTreasureState*, falls beim verlassen der Schatzposition noch transportierbarer Schatz vorhanden war. Wenn beim verlassen der Schatzposition jedoch die letzten Schatzeinheiten mitgenommen wurden, wechselt die Strategie in den Zustand *SearchTreasureState*. In Abbildung 31 ist ein Zustandsautomat mit den zugehörigen Zuständen und Übergängen dargestellt.

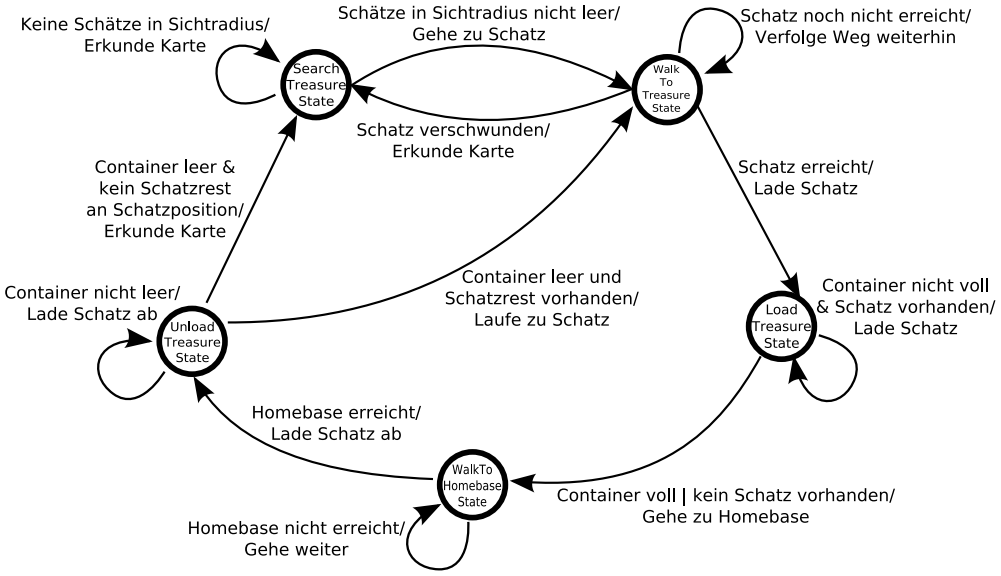
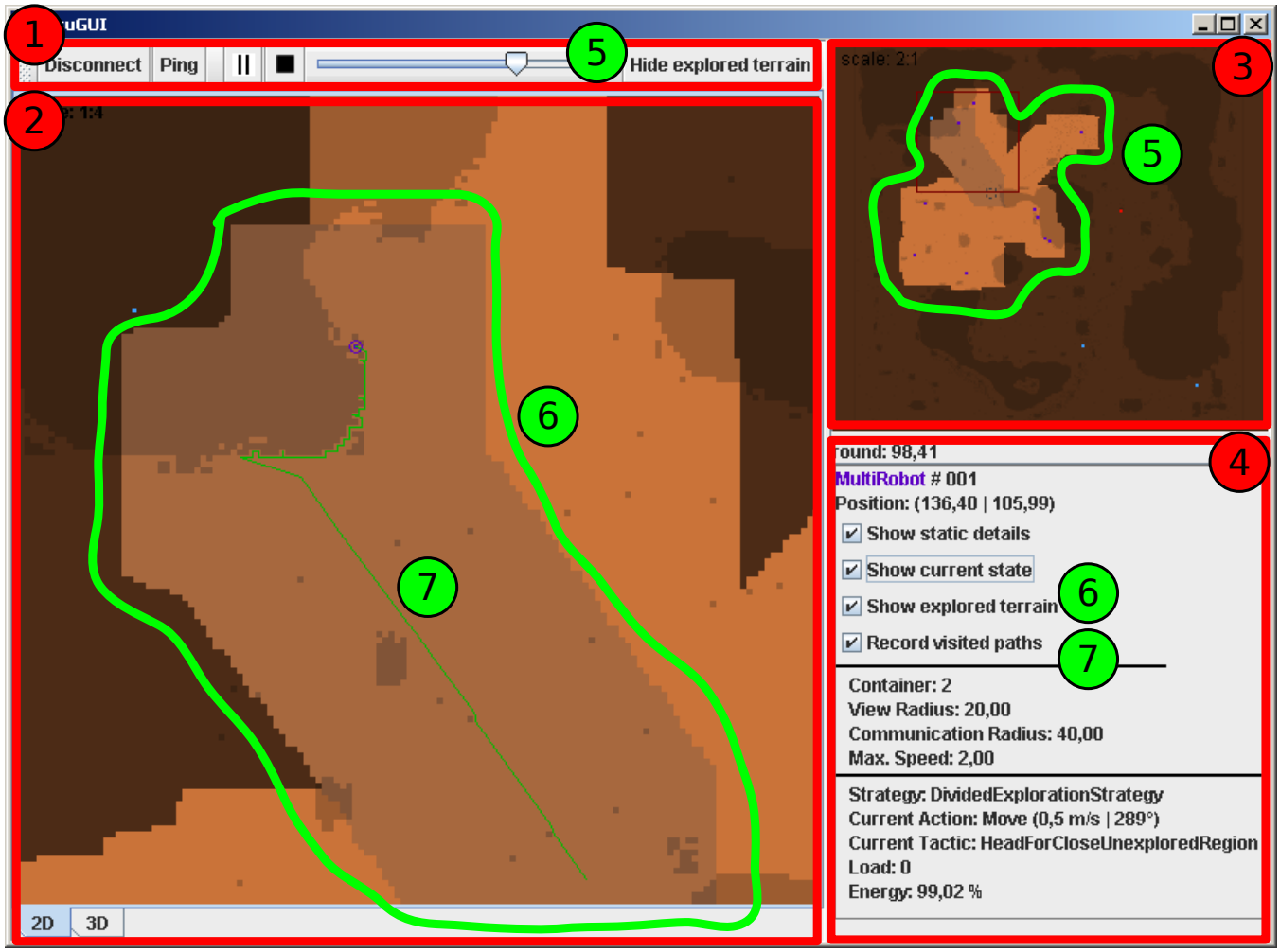


Abbildung 31: Zustamdsautomat von der Transporter-Strategie

5. Visualisierung

In diesem Kapitel beschreiben wir die dritte Hauptkomponente des Gesamtsystems, die Visualisierung und geben einen Überblick über ihre Bedienung, sowie die Architektur und innere Abläufe, ebenso wie über verwendete Datenstrukturen und Algorithmen.
Die Visualisierung besteht im Wesentlichen den vier Komponenten Menüleiste, Hauptansicht, Minikarte und Detailansicht und ist in Abbildung 32 dargestellt.



Menüleiste (1), Hauptansicht (2), Minikarte (3), Detailansicht (4), insgesamt explorierte Karte (5), von Roboter explorierte Karte (6), abgelaufene Wege (7)

Abbildung 32: Die Visualisierung

5.1. Architektur

Abbildung 33 zeigt einen Überblick Über den Aufbau der Visualisierungskomponente: Die Schnittstelle zum Kernel wird durch die Klasse *MainControl* repräsentiert. Mit der *MainControl* verbunden sind a) die GUI-Klassen der Vi-

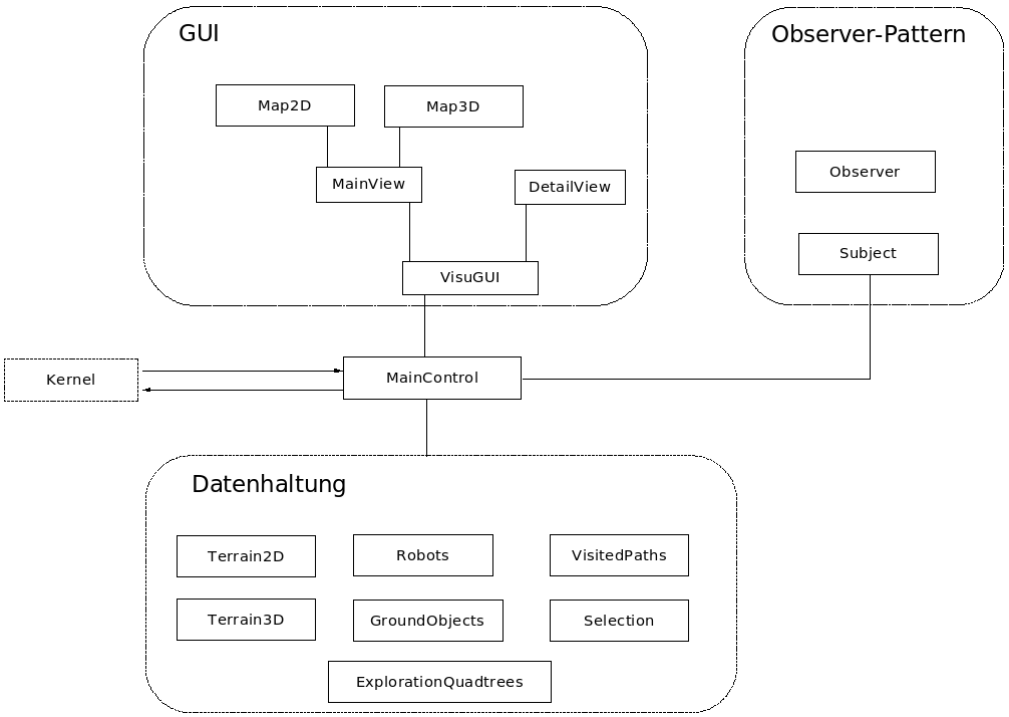


Abbildung 33: Grobarchitektur der Visualisierung

sualisierung, die für die User-Interaktionen und für die konkrete Visualisierung der Simulation verantwortlich sind, b) eine Reihe Datenstrukturen (insbesondere für die visualisierungsseitige Speicherung der Map und der Roboter) und c) einige weitere Hilfsklassen. All diese werden im nächsten Abschnitt 5.2 detailreicher beschrieben.
Als wichtiges Entwurfskonzept ist in der Architektur das Observer-Pattern (vgl. [GHJV95]) zu finden. Konkret implementiert wurde es in Klassen *Subject* und *Observer* sowie allen Unterklassen der *Observer*-Klasse. Mittels dieses Patterns wird dafür gesorgt, daß die Aktionen der Simulation in der Visualisierung einfach und effizient publik gemacht werden.

5.2. Wichtige Klassen

In diesem Abschnitt werfen wir einen detaillierteren Blick auf das Innere des Visualisierungsmoduls. Dabei stellen wir dar:

1. wie die Abläufe innerhalb der Visualisierung aussehen
2. wie die Datenhaltung aussieht
3. was die konkrete Ansicht auf die Simulation leistet
4. wie Features wie Detailansicht, Anzeige der explorierten Karte und der abgelaufenen Wege einzelner Roboter arbeiten.

5.2.1. Anbindung an den Kernel und Ablaufsteuerung

Wie schon in Kapitel 3.2.2 dargestellt, kommunizieren Kernel und Visualisierung über Nachrichten, die über Java-RMI verschickt werden. Visualisierungsseitig wird die Kommunikation von der Klasse *MainControl* abgewickelt. Sie ist ein Thread und das Gegenstück zur *Controller*-Klasse im Kernel; damit ist sie für das visualisierungsseitige Message-Handling verantwortlich: Sie behandelt einerseits die ankommenden Nachrichten und verarbeitet die in ihnen gekapselten Informationen über die Simulation, andererseits werden hier die von der Visualisierung ausgehenden Nachrichten erstellt und verschickt. Da die Anzeige der Visualisierung ausschließlich von den Nachrichten abhängt, die vom Kernel ankommen, ist die *MainControl* von zentraler Bedeutung. Zusätzlich wird in der *MainControl* die Visualisierung initialisiert – die anderen Hauptklassen der Visualisierung werden hier instanziiert.
Implementiert ist die *MainControl*-Klasse als Zustandsautomat. Sie verfügt über eine Reihe Zustände, die sich je nach Simulationszustand und nach Verbindungszustand zum Kernel ändern:

Zustand	**Beschreibung**
UNREGISTERED	Es besteht keine Verbindung zum Kernel.
REGISTERED	Eine Verbindung zum Kernel ist hergestellt, aber es läuft keine Simulation.
TERRAIN_REQUESTED	Der Kernel hat zum Initialisieren des Geländes aufgefordert, aber das Gelände wurde nicht gefunden und wird daher beim Kernel angefragt.
TERRAIN_INITIALIZED	Das Gelände wurde initialisiert, aber der Kernel hat noch nicht zum Initialisieren der Scene aufgefordert
SCENE_INITIALIZED	Das Gelände und die Scene wurden initialisiert, aber der Kernel hat noch keine Runde gestartet
SIMULATION_RUNNING	Die Simulation läuft.
SIMULATION_PAUSED	Die Simulation wurde pausiert.

Tabelle 3: Zustandsautomat der Klassse MainControl

Mittels des Zustandsautomaten ist auch die Fehlerbehandlung implementiert: Nachrichten vom Kernel werden nur immer in bestimmten Zuständen akzeptiert; kommen zum Zustand nicht passende Nachrichten an, werden diese nicht akzeptiert und es wird eine Fehlermeldung ausgegeben.

Ein weiteres wichtiges Konzept der Ablaufsteuerung im Visualisierungsmodul ist die Implementierung des Observer-Patterns, wie bereits in Abschnitt 5.1 kurz angesprochen wurde. Dieses Pattern sieht folgendes vor: Objekte, die über Änderungen im Kernel informiert werden sollen (dazu zählt insbesondere die konkrete Anzeige der Simulationsdaten, aber beispielsweise auch Datenhaltungsobjekte), registrieren sich beim *Subject*-Objekt, und zwar für eine oder mehrere von verschiedenen Update-Methoden. Diese werden bei bestimmten Events dann in allen registrierten Objekten aufgerufen und sind im einzelnen:

Zustand	**Beschreibung**
frameUpdate()	Wird jeden Frame aufgerufen
roundUpdate(newRoundMessage)	Wird jede Runde aufgerufen
sceneUpdate(initSceneMessage)	Wird aufgerufen, wenn eine neue Szene initialisiert wird
terrainUpdate(initTerrainMessage)	Wird aufgerufen, wenn ein neues Terrain initialisiert wird
robotDetailUpdate(robotDetailUpdateMessage)	Wird aufgerufen, wenn ein Roboter seine Daten ändert. Das umfaßt insbesondere auch Positionsänderungen des Roboters, was diese Methode sehr wichtig macht
groundObjectDetailUpdate(groundObjectDetailUpdateMessage)	Wird bei Änderungen von Bodenobjekten aufgerufen
simulationMessage(simulationPausedMessage)	Wird aufgerufen, wenn die Simulation pausiert wurde
simulationMessage(simulationResumedMessage)	Wird aufgerufen, wenn die Simulation fortgeführt wurde
simulationMessage(simulationStoppedMessage)	Wird aufgerufen, wenn die Simulation gestoppt wurde
simulationMessage(simulationSpeedChanged)	Wird aufgerufen, wenn kernelseitig die Simulationsgeschwindigkeit geändert wurde

Tabelle 4: Update-Methoden für die Ablaufsteuerung im Visualisierungsmodul

5.2.2. Datenhaltung

Für die Datenhaltung im Visualisierungsmodul sind im wesentlichen die Klassen *Robots*, *GroundObjects*, *Selection*, *ExplorationQuadtrees* und *VisitedPaths*

verantwortlich; dabei werden die beiden letzteren im Abschnitt 5.2.5 erläutert. Die Klasse *Robots* besteht im wesentlichen aus einem Array sämtlicher Roboter und einigen Methoden auf dieser Liste. Die wichtigsten sind hier kurz erläutert:

1. `public synchronized void init(RobotInit robotInit)`: Initialisiert die Roboter anhand der übergebenen *RobotInit*-Nachricht
2. `public synchronized Set<Robot> getRobotsInRegion(Rectangle2D region)`: Gibt die Roboter innerhalb der spezifizierten Region zurück
3. `public synchronized void updateRobots(RobotChanges robotChanges)`: Aktualisiert die Roboter anhand der übergebenen *RobotChanges*-Nachricht

Die Klasse *GroundObjects* ist analog aufgebaut: Sie besteht in erster Linie aus einer Menge von *GroundObject*-Objekten und einigen Methoden auf dieser Menge; die wichtigsten davon sind:

1. `public synchronized void init(GroundObjectInit groundObjectInit)`: Initialisiert die Bodenobjekte anhand der übergebenen *GroundObjectInit*-Nachricht
2. `public synchronized Set<GroundObject> getGroundObjectsInRegion(Rectangle2D region)`: Gibt die Bodenobjekte innerhalb der spezifizierten Region zurück
3. `public synchronized void updateGroundObjects(GroundObjectChanges groundObjectChanges)`: Aktualisiert die Bodenobjekte anhand der übergebenen *GroundObjectChanges*-Nachricht

In der *Selection*-Klasse schließlich werden die Objekte verwaltet, die der Benutzer ausgewählt hat. Letztlich stellt sie eine Datenstrukur ähnlich einer normalen Liste dar, nur sind die Zugriffsoperationen stark modifiziert. Anforderungen an diese Auswahl – wie beispielsweise die Notwendigkeit, daß bei Auswahl eines Roboters die Detailansicht aktualisiert wird – werden in diesen Zugriffsoperationen erfüllt. Ihre wichtigsten Methoden sind:

1. `public final synchronized void select(Set<? extends Selectable> selection)`: Wählt Objekte aus. Möglich sind Objekte, die das Interface *Selectable* implementieren; das sind beim aktuellen Stand des Projektes *Robot* und *Groundobject*.
2. `public final synchronized void add(Set<? extends Selectable> selection)`: Die übergebenen Selectables werden der Selektionsliste hinzugefügt.
3. `public final synchronized void remove(Set<? extends Selectable> selection)`: Die übergebenen Selectables werden aus der Selektionsliste entfernt.
4. `public final synchronized void reset()`: Löscht die gesamte Liste von Selectables

Bei all diesen Methoden wird sichergestellt, daß bei Selektion von Robotern oder Bodenobjekten die Anfragen an den Kernel über Detailaktualisierung gestartet und bei Deselektion gestoppt wird.

5.2.3. 2D-Visualisierung

Die 2D-Visualisierung besteht im Wesentlichen aus der *Map2D* und dem *Terrain2D*. Die *Map2D* übernimmt dabei das zeichnen des Geländes, der Roboter und der Bodenobjekte und bietet Funktionen wie Zoomen und Verschieben per Maussteuerung. Das *Terrain2D* ist die zugehörige Datenhaltungsklasse, welche die Bereichsanfragen auf dem im *TerrainQuadtree* gespeicherten Gelände übernimmt. Für eine bessere Bildqualität werden in inneren Knoten die zugehörigen Farben der Kinder interpoliert.

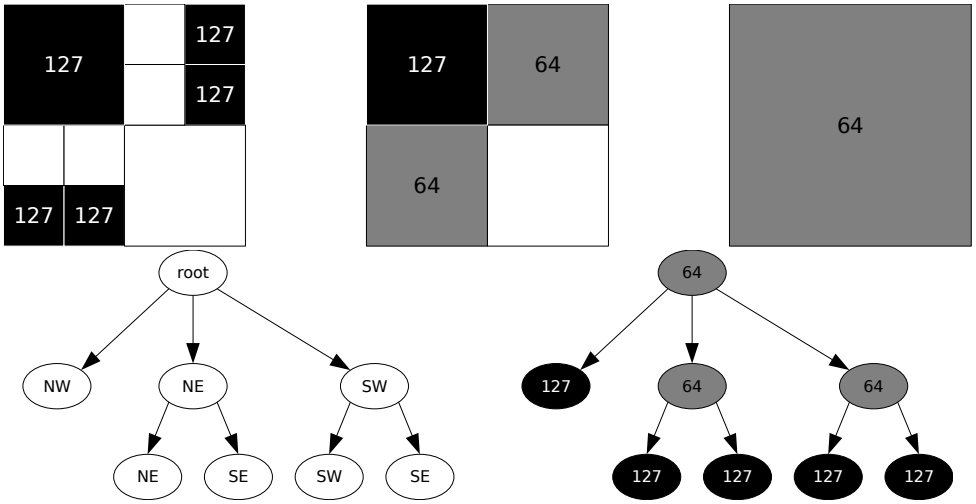


Abbildung 34: Repräsentation der Karte im TerrainQuatree in unterschiedlichen Detailgraden. In gröberen Detailgraden werden die Werte der Kinder interpoliert, um auch feine Strukturen bei kleinen Zoomstufen angenähert darstellen zu können.

Abbildung 35: Resultat des Geländezeichnens ohne und mit Interpolation.

5.2.4. Detailansicht

5.2.5. Erkundete Karte und abgelaufene Wege

Die 2D-Visualisierung ist dazu in der Lage, die erkundete Karte einzelner sowie aller Roboter anzuzeigen. Dazu wird, wenn dieses erstmalig vom User angefordert wird, der Quadtree der erkundeten Karte vom Roboter über den Kernel zur Visualisierung geschickt und dort in einer eigenen Datenstruktur (zu finden in der Klasse *ExplorationQuadtrees*) gespeichert. Weil das sinnvollerweise jede Runde, die die explorierte Karte angezeigt werden soll, geschehen müßte, damit die angezeigte explorierte Karte immer auf dem neuesten Stand ist, wäre die Netzwerklast sehr schnell sehr hoch. Also werden die einzelnen Quadtrees visuseitig nicht nur gespeichert, sondern auch in jeder neuen Runde aktualisiert, indem der Quadtree um das vom Roboter gesehene Gebiet erweitert wird.
Zusätzlich bietet die 2D-Visualisierung die Möglichkeit, die abgelaufenen Wege einzelner Roboter zu speichern und in die Karte zu zeichnen. Damit die Speicherung dieser abgelaufenen Wege (findet statt in der Klasse *VisitedPaths*) nicht zu platzbedürftig wird, ist dieses Feature nur für einzelne Roboter implementiert – es ist also nicht möglich, sich auf Knopfdruck sämtliche abgelaufene Wege anzeigen zu lassen. Des weiteren wurde mittels eines simplen Tricks der Speicherbedarf zwar nicht minimiert, aber immerhin leistbar gemacht: Anstatt jeden einzelnen Punkt zu speichern, den der Roboter besucht hat, werden nur diejenigen gespeichert, die der Roboter bei einer Richtungsänderung besucht. Zwischen diesen Punkten werden dann Linien gezeichnet. Diese Lösung ist simpel, aber praktikabel – dennoch gibt es zweifellos geeignetere Speicherungsmöglichkeiten, die zu implementieren allerdings aus Zeitgründen nicht möglich war.

5.2.6. 3D-Visualisierung

In der 3D-Ansicht wird die Szene hübsch aufbereitet, damit man auch mal was nett anzusehendes zeigen kann. Alle Informationen werden dabei aus den 2D-Daten extrahiert und ggf um eine Höheninformation ergänzt (siehe Abschnitt 5.3.2). Es ist also kein echtes 3D – man könnte es 2.5D nennen.
Im vorletzten Satz steckt allerdings ein wichtiges Detail: es sind schon die X- und Y-Werte gegeben. In der 2D-Ansicht gehen positive X-Werte nach rechts und positive Y-Werte nach unten. Die Hindernisse würden aus dem Bildschirm rauskommen. Hätten die Hindernisse positive Z-Werte, so bekäme man ein linkshändiges Koordinatensystem – OpenGL erwartet aber ein rechtshändiges. Um das nicht zur Laufzeit immer umrechnen zu müssen, haben wir folgende etwas unintuitive aber notwendige Entscheidung getroffen:

X in 2D entspricht X in 3D
Y in 2D entspricht Y in 3D
Z in 3D ist negativ!

Das Rendering ist mit der *Map3D* in der Visualisierung verankert. Die wichtigsten Klassen und Konzepte sind stecken aber in den folgenden Punkten:

Rendering des Geländes Im Preprocessing (siehe Abschnitt 5.3.2) wird das Gelände in größere Hindernisse (Dreiecke) und Steine (sich wiederholende Instanzen einiger weniger 3D-Stein-Modelle) unterteilt und in einen Octree einsortiert. Zur Laufzeit werden diese Daten aus der Klasse *TerrainOctree* ausgelesen. Diese Klasse ist *bewusst* nur aus primitiven Datentypen statt Objekten zusammengesetzt, da sich gezeigt hat, dass mit Objekten (z.B. ArrayList statt Array oder für jedes Dreieck ein Objekt, o.ä.) der Speicherbedarf sehr stark ansteigt. Genutzt werden diese Daten vom *TerrainRenderer*. Dieser durchläuft den *TerrainOctree* in preorder und rendert alle Dreiecke und Steine, die in den durchlaufenen Knoten gespeichert sind. Da das Gelände deutlich mehr Dreiecke enthält, als die Grafikkarte zeichnen kann, darf nur eine kleine Menge der Dreiecke gezeichnet werden. Daher wird vor dem rekursiven Abstieg in einen Teilbaum überprüft, ob es sich überhaupt lohnt, diesen zu zeichnen. Es gibt 3 Gründe, warum ein Teilbaum verworfen werden kann:

1. die BoundingBox des Teilbaums ist weit vom Betrachter weg und so klein, dass in ihm enthaltenen Dreiecke vermutlich kaum sichtbar wären
2. die BoundingBox des Teilbaums ist nicht sichtbar (Frustum-Clipping)
3. das Rendern des Frames überschreitet ein bestimmtes Zeitlimit

Zu Punkt 1: Es wird zu jedem Knoten die Größe seiner BoundingBox und sein Abstand zur NearPlane (Zeichenebene) ins Verhältnis gesetzt. Wird dabei ein bestimmtes Limit nicht erreicht, so wird der komplette Teilbaum verworfen. Um gleichzeitig die Bildqualität so hoch wie möglich zu halten und dennoch eine flüssige Navigation zu gewährleisten, wird dieses Limit automatisch angepasst, je nachdem, wie lange das Rendern des Frames gedauert hat.
Zu Punkt 3: Derzeit ist es so eingestellt, dass das Rendern eines Frames nach 200ms angebrochen wird, um mindestens 5 FPS zu sicherzustellen. Um die Störungen im Bild zu minimieren, die durch ein plötzliches Abbrechen entstehen können, werden die Kinder des Knotens in near-to-far-order durchlaufen, so dass eher weiter Weg liegende liegende Dinge wegfallen, als nahe am Betrachter liegende.

Texturierung des Geländes (Shader) Da die Grafikkarte (bzw. der Bus) mit dem Übertragen der Vertex- und Normalendaten schon genug zu tun hat, wird auf die Übermittlung von Texturkoordinaten verzichtet. Statt dessen übernimmt der *MonumentShader* die Texturierung per Shaderprogramm. Dazu wird eine Oberflächentextur anhand der xy-Koordinaten von oben auf die Hindernisse projiziert und eine Höhentextur anhand der z-Koordinate von der Seite an die Hindernisse projiziert. Die Shaderprogramme sind in GLSL[3D] geschrieben und sind so einfach, dass sie keiner Erklärung bedürfen.
Kann das Shaderprogramm nicht geladen werden (z.B. weil die Grafikkarte zu alt ist), so wird automatisch eine optisch nicht ganz so schöne Methode von OpenGL verwendet, die ohne Shaderprogramme auskommt (Fallbackmodus). Dieser Fallbackmodus verwendet *glTexGen* zur Erzeugung der Texturkoordinaten und texturiert nur mit der Höhentextur.

Roboter, Schätze und Flags Die Roboter, die Schätze und die Flags liegen als .obj-Datei vor und werden mit Hilfe des *OBJLoaders* beim Start der Visualisierung geladen. Da die Roboter aus bis zu 917 Polygonen bestehen, wurden zusätzlich zu den Robotern verschiedene Level-Of-Detail Stufen modelliert, die in Abhängigkeit der Distanz zwischen Roboter und Kamera gewählt werden. Abbildung 36 zeigt die fünf Robotertypen.

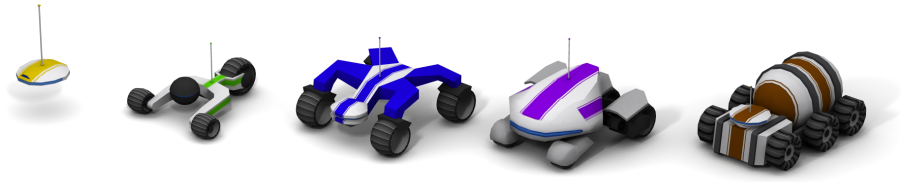


Abbildung 36: Die Robotertypen: Relais, Explorer, Worker, Multirobot und Transporter (von links nach rechts).

Kamera und Userinteraktion Die Klasse *CameraAndMouse* ist für alles zuständig, was mit der Position der Kamera in der Szene und deren Beeinflussung durch den Benutzer zu tun hat. Zu allererst ist das die Position und Blickrichtung der Kamera. Die Position ist ein einfacher Vektor, die Blickrichtung wird durch einen links-rechts-Winkel und einen hoch-runter-Winkel angegeben (siehe Abbildung 37). Dies ermöglicht eine einfache Manipulation durch die Maus.

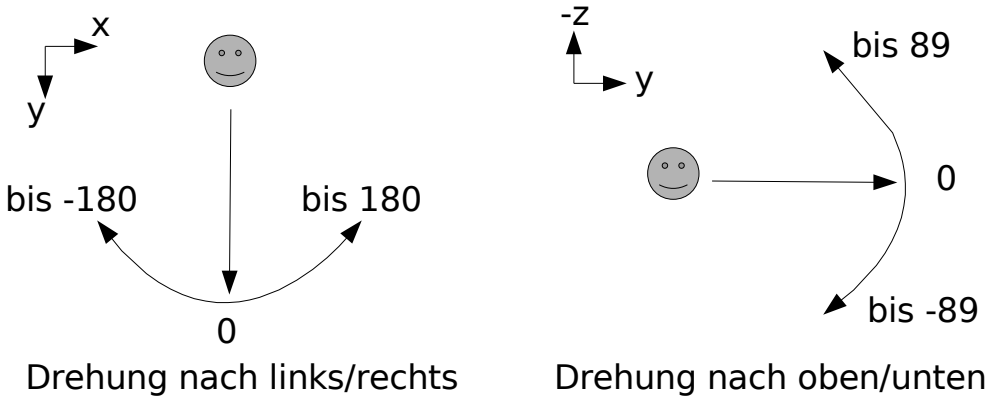


Abbildung 37: Die Blickrichtung der Kamera

Die zweite Aufgabe ist das Bereitstellen aller notwendigen Operationen für das Frustum-Clipping. Immer wenn sich die Position oder Blickrichtung ändert, werden die 6 Ebenen gespeichert, welche das Frustum beschränken. Anhand dieser kann dann getestet werden, ob sich ein Würfel im Frustum befindet. Die Implementierung richtet sich exakt nach zwei Tutorials[Mor, Hol] und ist dort ausführlich erklärt. Hier wäre sicherlich eine Implementierung mit Objekten (3D-Vektoren, ...) deutlich schöner zu lesen, hat aber zu spürbaren Geschwindigkeitseinbußen geführt.

Die dritte wichtige Aufgabe ist die Userinteraktion. Hier sollte eigentlich alles selbsterklärend (bzw. durch JavaDoc ausreichend erklärt) sein. Einzig zum Tastaturlistener ist noch zu erwähnen, dass hierfür ein eigener Thread eigerichtet wurde, der regelmäßig schaut, welche Tasten gedrückt sind, und dementsprechend die Kamera verändert. Dies war notwendig, da die von Java ausgelösten Events beim Halten einer Taste zu langsam und selten kommen und somit die ganze Navigation sehr ruckelnd wirkte.

5.3. Preprocessing

Im Kernel reicht es, das Gelände in "Hindernis" und "kein Hindernis" einzuteilen. Für eine ansprechende Darstellung reicht dies nicht aus. Für die 2D-Visualisierung müssen die Daten so aufbereitet werden, dass auch die ganze Karte (wenn stark herausgezoomt wird) flüssig gezeichnet werden kann – dazu müssen größere Flächen durch einzelne Grauwerte approximiert werden. Für die 3D-Visualisierung müssen aus den 2D-Daten zunächst dreidimensionale Hindernisse erzeugt werden und diese anschließend in einer Datenstruktur gespeichert werden, die das flüssige Rendern erlaubt.
Da dieser Prozess speicher- und zeitaufwendig ist, findet er einmalig statt. Da im Preprocessing Zeit nicht so eine große Rolle spielt, wurde eher auf Speicherverbrauch als auf Laufzeit optimiert. Die resultierenden Datenstrukturen werden in Dateien geschrieben. So können die vorverarbeiteten Daten ohne lange Wartezeit in die Visualisierung geladen werden.
Der *TerrainForVisuConverter* ist für das Preprocessing zuständig. Er wird vom Editor oder per Kommandozeile aufgerufen und erhält eine XML-Geländebeschreibung. Er delegiert die Erzeugung der 2D- bzw. 3D-Daten an das *Terrain2D* bzw. *Terrain3D*. Sind diese mit dem Preprocessing fertig, werden sie in eine Datei serialisiert, welche im selben Ordner wie die XML-Datei liegt und deren Namen sich nur durch die neue Endung `v2d` bzw. `v3d` unterscheidet.

5.3.1. 2D-Gelände

Der Einstiegspunkt zum Preprocessing des 2D-Geländes ist der Konstruktor *Terrain2D(InputStream regionXml)*. Dieser bekommt die XML-Beschreibung, parst ein paar Metadaten und ruft dann die Methode *fromXML* des *TerrainQuadtree* auf, welcher die eigentliche Arbeit erledigt.
Die Knoten des *TerrainQuadtree* werden bottom-up anhand der XML-Baumstruktur erzeugt. Als einzige zusätzliche Berechnung muss der Grauwert eines Knotens berechnet werden. Dazu wird die schwarze Fläche der Kindsknoten (gemessen in Quadtraten der Größe 1×1-Meter) aufsummiert und durch die gesamte Fläche des Quadtreeknotens geteilt.

5.3.2. 3D-Gelände

Der Einstiegspunkt zum Preprocessing des 3D-Geländes ist der Konstruktor *Terrain3D(Terrain2D terrain2d, ...)*. Hier laufen folgende Schritte ab:

- Übernehmen einiger Metadaten aus dem *Terrain2D*

- Erzeugung der 3D-Hindernisse mittels *QuadtreeMonumenter* – Ausgabe: Dreiecke und Steine als *TerrainData3D*
- Konvertierung der *TerrainData3D* in Listen aus *PreprocessingTriangle*s und *PreprocessingStone*s
- Einsortierung dieser Daten in einen *PreprocessingTerrainOctree*
- Konvertierung des *PreprocessingTerrainOctree* in den speicherplatzoptimierten *TerrainOctree*

Beachte: in (fast) der ganzen Visualisierung ist die Z-Komponente der 3D-Vektoren ≤ 0 (Grund: siehe Abschnitt 5.2.6 "3D-Visualisierung"). Genauer: im QuadtreeMonumenter werden zunächst noch positive Z-Werte erzeugt (historisch bedingt) – am Ende werden die Z-Werte dann negativ gemacht und bleiben ab da negativ.

QuadtreeMonumenter Das Modell der Simulation ist auf eine zweidimensionale Beschreibung des Geländes beschränkt. Um dennoch Hindernisse in der 3D Visualisierung nicht flach darzustellen, war es unsere Aufgabe aus beliebig geformten Hindernissen eine ansprechende Berglandschaft zu generieren. Da die Formen von Hindernissen auch sehr untypisch für Berge sein können, fiel unsere Wahl auf Tafelberge, da diese auch bei sehr scharfen Konturen und abstrakten Formen von Hindernissen ein einigermaßen realistisches Gesamtbild schaffen. Nichtsdestotrotz gibt es ein Flag, dass optional die den Generator auf Pyramiden bzw. Hügelartige Berge setzt.
Der *QuadtreeMonumenter* bekommt das *Terrain2D* übergeben, generiert daraus ein Netz aus Dreiecken und gibt dieses als *TerrainData3D* zurück. Er erzeugt dabei drei Arten von Hindernissen: Steine, Schutthügel und Monuments. Dabei geht er wie folgt vor:

1. Übersetze den Quadtree aus *Terrain2D* in einen *MeshQuadtree*, der einige zusätzliche Funktionen (Nachbarschaftsuchen, Zusammenhangskomponenten finden, u.ä.) und die Möglichkeit Vertexindices zu speichern bietet.
2. Trenne 1m×1m Hindernisse aus dem Quadtree heraus und speichere ihre Koordinaten in einer Liste. Hieraus werden die Steine.
3. Trenne große Zusammenhangskompenten über einem gewissen Threshhold (default: 32m) heraus und speichere Sie in einem zweiten Quadtree (siehe Abbildung 38). Hieraus werden die Monuments. Der übriggebliebene Debrisquadtree beschreibt die Schutthügel.
4. Löse bei beiden Quadtrees die Ecken von Hindernissen auf 0,5m×0,5m auf. Unterteile weiterhin solche Randblätter, die die maximal angegebene Randgröße überschreiten, solche die mit mehr als einer Seite am Rand liegen, und solche von denen mindestens eine Seite nur teilweise am Rand liegt (siehe Abbildung 39).

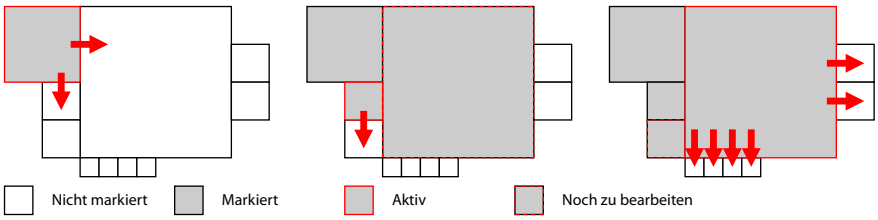


Abbildung 38: Zusammenhängende Hindernisstrukturen finden

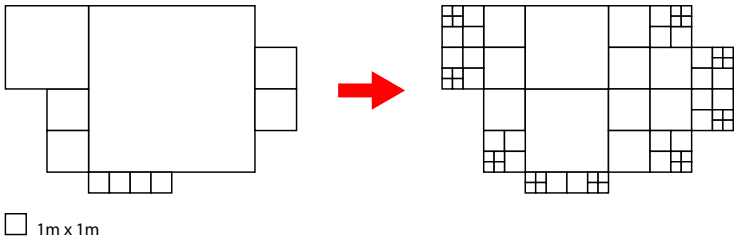


Abbildung 39: Hindernisstrukturen an den Ecken höher auflösen

5. Balanciere den Quadtree, d.h. sorge durch Unterteilen dafür, dass die Größen zweier benachbarter Blätter sich nie um mehr als den Faktor zwei unterscheiden (siehe Abbildung 40).

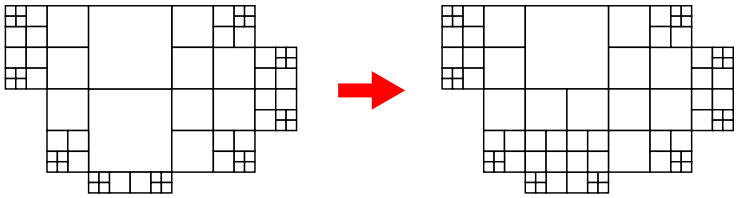


Abbildung 40: Quadtree balancieren

6. Berechne bei beiden Quadtrees für alle Eckpunkte der Blätter die Cityblockabstände bis zum Rand des Hindernisses (siehe Abbildung 41). Wenn noch nicht vorhanden, füge Vertices mit dem Randabstand als z-Wert an den Ecken der Quadtreeblätter ein. Die xy-Koordinate ergibt sich aus dem Quadtreeblatt. Merke dir die Indices der Vertices und speichere sie. Speichere sie ebenfalls in den Nachbarn, die diese Vertices teilen.

7. Berechne für den Debrisquadtree und den Monumentquadtree mit der zugehörigen Höhenfunktion (Eingabe Randabstand, Ausgabe Höhe) die endgültige Höhe der Vertices. Füge einen Mittelpunkt in jedes Blatt ein und interpoliere dessen Höhenwert.

8. Berechne Normalen einer Kugel vor und ordne Ihnen Indizes zu.

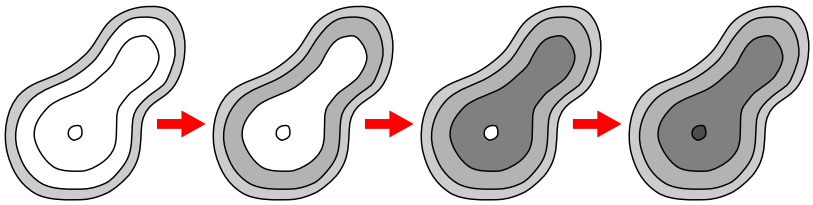


Abbildung 41: Iterativ Abstände zum Rand berechnen

9. Erzeuge aus jedem Blatt beider Quadtrees Dreiecke (siehe Abbildung 42), die auf die Vertex-Indices referenzieren und berechne die zugehörigen Normalenindizes.

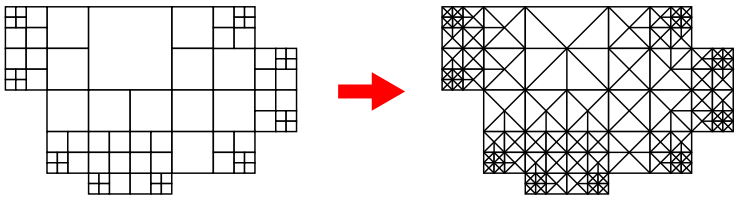


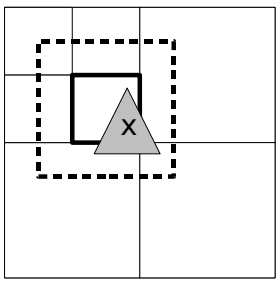
Abbildung 42: Dreiecke generieren

10. Ordne jedem Eintrag aus der Steinliste einen zufälligen Index zu, der auf einen der zehn vorgefertigten stone-Objekte verweist.

PreprocessingTerrainOctree Der Einstiegspunkt zum Einsortieren der Dreiecke und Steine in den Octree die statische Methode *preprocessTerrainOctree* der Klasse *PreprocessingTerrainOctree*. Dieser bekommt neben den Dreiecks- und Steindaten einige Parameter welche in den folgenden Absätzen bei ihrem jeweiligen Einsatzpunkt erklärt werden. Dabei wird wie folgt vorgegangen:

1. *Füge die Dreiecke ein.* Das ganze funktioniert nach dem Prinzip des "Loose Octree" [Fis07, Tha00]. Grob gesagt: die Boundingboxen der Octree-Knoten werden in jeder Dimension doppelt so groß wie in einem gewöhnlichen Octree (sie überlappen sich also). Die Tiefe, in der ein Dreieck eingefügt wird, hängt damit nur noch von der Größe seiner Boundingbox ab und die Position im Octree nur vom Mittelpunkt seiner Boundingbox. Großer Vorteil: es können keine kleinen Dreiecke mehr "oben im Octree hängen bleiben", nur weil sie ungünstig liegen (siehe Abbildung 43).
   Zusätzlich gibt es einen Parameter *maxDepth*, der bestimmt, wie tief der Baum maximal werden darf. Dieser Parameter ist relativ zu log(Breite des Geländes) zu verstehen, d.h. die wirkliche Maximaltiefe ist log(Breite des Geländes)+ *maxDepth*. Der Grund: so ist z.B. 0 immer ein sinnvoller Wert, unabhängig von der Kartengröße.

2. *Füge die Steine ein.* Die Steine werden grundsätzlich in der Tiefe log(Breite des Geländes) eingefügt. Es wird immer rekursiv in den Kindknoten hinabgestiegen, in



Aufgrund seiner Größe muss das Dreieck in die BoundingBox (gestrichelt) eines Knotens der Tiefe 2 passen. In welchem Knoten (dicke Linie) das Dreieck landet, bestimmt nur sein Mittelpunkte (x).

Abbildung 43: Einfügen eines kleines Dreiecks in einen Loose Octree

dem der Mittelpunkt des Steins liegt. Grund für die Tiefe: in einem normalen (nicht-loose) Octree würde dann der Stein mit Größe $1 \times 1 \times 1$-Meter exakt in die Boundingbox dieses Octreeknotens passen.
Der Mittelpunkt dieses Octreeknotens entspricht dann genau der Position des Steins. Dies ist wichtig, da im *TerrainOctree* um Platz zu sparen nur noch die Information gespeichert wird, ob ein Stein in einem Knoten vorhanden ist, nicht aber seine Position. Diese ist nur noch implizit durch den Mittelpunkt des Octreeknotens gegeben.

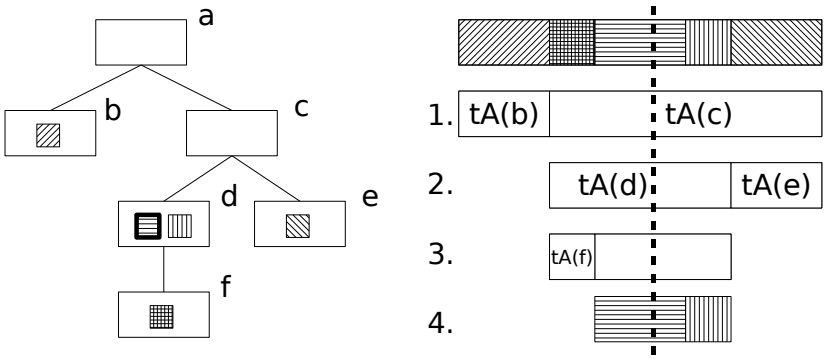
3. *Ziehe zufällig ausgewählte Dreiecke im Octree nach oben.* Dies folgt dem Konzept des "Randomized Sample Trees" [KKF+04]. Die Implementierung der Methode *pullUpSamples* ist eine direkte Umsetzung des Algorithmus aus dem Paper. Dort gibt es auch schöne Illustrationen, weswegen hier darauf verzichtet wird. Es gibt zwischen Implementierung und Paper folgende Entsprechungen:

Implementierung	Paper
pullUpSamples	createSample
sampleTreeQuality	c
areaQuotient	q_u
numSamples	m_u
samples	M
trianglesArea	$A(P(u))$
pullUpSample	"choose polygon ... with probability ..."

Tabelle 5: Namensunterschiede von Methoden zwischen Implementierung und Paper

Dabei bedarf die Methode *pullUpSample* noch einiger Erläuterungen: Um aus einer Menge von Dreiecken zufällig eines gewichtet nach Flächeninhalt auszuwählen, kann man jedem Dreieck ein Intervall zwischen 0 und dem Gesamtflächeninhalt zuweisen, welches seiner Größe entspricht und sich mit keinem anderen Intervall überschneidet. Nun wählt man einen zufälligen

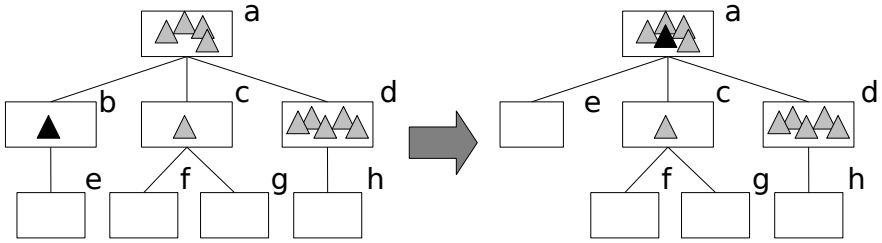
Wert zwischen 0 und dem Gesamtflächeninhalt. Das Dreieck, in dessem Intervall dieser Wert liegt, ist damit ausgewählt.
Eine Einteilung in solche Intervalle kann man z.B. vornehmen, indem man die Dreiecke so sortiert, wie sie bei einem Baumdurchlauf post-order auftreten würden. Dann ist $[a_i, b_i[$ das Intervall des i-ten Dreiecks mit Flächeninhalt $area_i$, wobei $a_i := b_{i-1}$ und $b_i := a_i + area_i$.
Natürlich ist es unpraktikabel, für jede Dreiecksauswahl einen solchen Baumdurchlauf zu machen — es geht aber auch viel einfacher. Dazu muss in jedem Knoten gespeichert werden, wie groß der Gesamtflächeninhalt aller Dreiecke ist (*triangleArea*), die in ihm oder seinen Nachkommen gespeichert sind. Dann kann man das gesuchte Dreieck in $\mathcal{O}(\log(d) + k)$ finden (bei Baumtiefe d und k Dreiecken im Knoten, der das gesuchte Dreieck enthält). Wie das funktioniert, illustriert Abbildung 45.



Gegeben ist ein Baum, der 5 Dreiecke mit den Flächeninhalten 1 bzw. 2 enthält. Der Gesamtflächeninhalt ist 8. Es soll das Dreieck ausgewählt werden, welches zu der zufälligen Zahl $4.3 \in [0, 8[$ gehört. Der oberste Kasten zeigt die Sortierung der Dreiecke entsprechend des post-order Durchlaufs und mit Gewichtung nach Flächeninhalt. Hier sieht man, dass das horizontal linierte Dreieck aus Knoten d ausgewählt würde. Der Algorithmus ermittelt dies wie folgt:
Im 1. und 2. Schritt errechnet man anhand der *trianglesArea*-Werte der Kinder, in welchem Kind das Dreieck liegen muss. Im 3. Schritt sieht man an den Werten, dass das Dreieck in keinem Kind liegen kann (also in d selbst). Im 4. Schritt wird dann die Dreiecksliste von d linear nach dem passenden Dreieck durchsucht.

Abbildung 44: Beispiel für *pullUpSamples*

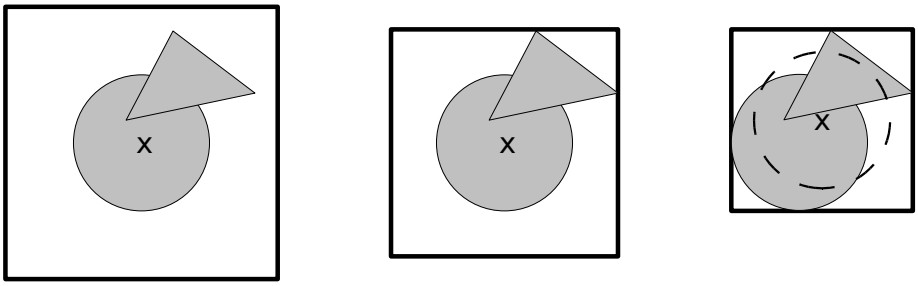
4. *Versuche wenig nützliche Octreeknoten einzusparen.* Knoten, die nur ein Kind haben und wenige Dreiecke beinhalten werden nach Möglichkeit eliminiert, denn sie machen den Octree unötig groß. Im Elternknoten wird dann die Referenz auf diesen Knoten durch sein einziges Kind ersetzt. Die Dreiecke des Knotens werden in den Elternknoten verschoben.
Der Parameter *triangleTreshold* ist eine Obergrenze für die Anzahl der Dreiecke, die im Elternknoten nach dieser Optimierung enthalten sein dürfen. Ist *triangleTreshold == 0*, dann wird diese Optimierung nur dann vorgenommen, wenn der Knoten keine Dreiecke enthält.



Sei der *triangleTreshold* == *5*. *c* kann nicht entfernt werden, da er mehr als ein Kind enthält. *d* kann nicht ersetzt werden, da *a* nachher $4 + 5 > triangleTreshold$ Dreiecke enthalten würde. *b* erfüllt alle Kriterien und kann daher entfernt werden.

Abbildung 45: Beispiel für *compressTreeStructure*

5. *Mache die Boundingboxen so klein wie möglich.* Dies dient dazu, das Frustumculling zu verbessern. Je passgenauer die Boundingbox, desto seltener kommt es vor, dass beim Frustumculling ein Octreeknoten im Frustum liegt, obwohl sein Inhalt nicht sichtbar ist.
   Traversiere dazu rekursiv und bottom-up den gesamten Baum. Setze Größe und Positionen der BoundingBoxen eines Octreeknotens so, dass die Größe minimal ist, aber dennoch sämtliche in diesem Knoten gespeicherten Dreiecke sowie die Boundingboxen aller Kinder noch hinein passen.
   Ein wichtiger Spezialfall liegt vor, wenn ein Knoten einen Stein enthält. Dann darf die Boundingbox nur verkleinert, aber nicht verschoben werden (Grund: siehe Punkt 2 – Illustration: siehe Abbildung 46).
   Der Parameter *shrinkBoundingBoxes* bestimmt, ob diese Optimierung ausgeführt werden soll, oder nicht.



links *Ausgangssituation:* ein Dreieck, ein Stein mit Durchmesser 1 und die BoundingBox mit Durchmesser 2.

mittig *Schrumpfen ohne Verschiebung:* dies ist die korrekte Situation, der Stein liegt nach wie vor im Mittelpunkt.

rechts *Schrumpfen mit Verschiebung:* das darf nicht passieren! Der Stein läge nicht mehr im Mittelpunkt und würde verschoben dargestellt (gestrichelter Kreis). Wäre an Stelle des Steins ein gleichgroßes Dreieck, so wäre dies das korrekte Resultat der *shrinkBoundingBoxes*-Methode.

Abbildung 46: Verschieben des Mittelpunktes bei *shrinkBoundingBoxes*

5.3.3. Ein paar Zahlen...

Gemessen auf einem Intel Core2 @ 2.66GHz mit 4GB RAM und "-Xmx2500G".

Karte	512M	moreReal	leopard	terr8192	benchm16km
Quadtree-Knoten	840	17.178	127.314	55.240	1.190.776
Quadtree-Blätter	330	9.046	75.596	21.230	504.058
Preprocessing-Zeit	6,6 s	9,9 s	27 s	94 s	510 s
Preprocessing-RAM	4,5 M	11 M	75 M	335 M	1.540 M
Octree-Knoten	5.037	19.884	138.547	750.426	3.254.970
Dreiecke	28.335	113.954	944.367	3.988.950	19.359.238
Steine	39	837	1.013	395	126.165
Ladezeit	0,7 s	0,8 s	5,6 s	23 s	128 s
RAM	2,5 M	5,2 M	35 M	145 M	770 M

Karte	16_10000	16_15000	16_30000	16_100000	terr100km
Quadtree-Knoten	578.157	861.736	1.687.942	5.369.011	4.947.243
Quadtree-Blätter	295.419	443.029	877.994	2.862.177	3.752.398
Preprocessing-Zeit	67 s	101 s	212 s	—	—
Preprocessing-RAM	305 M	430 M	830 M	out of mem	bei 2,5 G
Octree-Knoten	555.053	830.613	1.647.228	—	—
Dreiecke	2.567.952	3.845.391	7.642.939	—	—
Steine	0	0	0	—	—
Ladezeit	20 s	29 s	58 s	—	—
RAM	120 M	175 M	347 M	—	—

Tabelle 6: Testergebniss der Simulation mit unterschiedlichen Karten

6. Editor

Die Aufgabe bestand darin eine übersichtliche und benutzerfreundliche Oberfläche zusammen zu stellen, die die Erstellung und Bearbeitung von Karten ermöglicht.

6.1. Architektur

In diesem Kapitel wird der Aufbaue der Oberfläche (im folgenden GUI, engl. „Graphical User Interface“) näher erläutert. Informationen zur Bedienung des Editors finden Sie im Benutzerhandbuch (Menü “Help“). Die wesentlichen Komponenten der Klasse GUI sind in Abbildung 47 zu sehen.

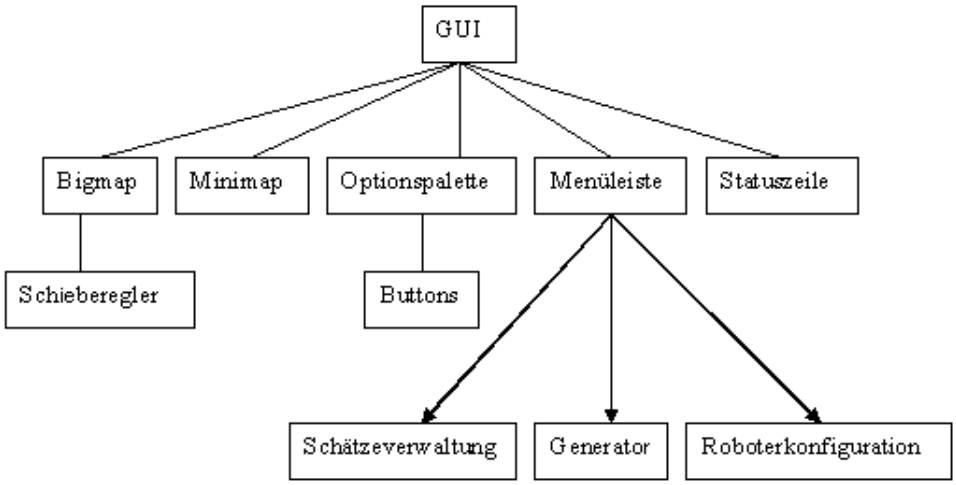


Abbildung 47: Grober Aufbau der GUI

6.1.1. Die Bigmap

Die Bigmap besteht aus einem JPanel und zwei Schiebereglern, eines für die horizontale und eines für die vertikale Schieberichtung.
In der Bigmap wird ein Ausschnitt der Karte in skalierter Form angezeigt. Dafür werden nur die Hindernisse aus dem Quadtree ausgelesen und gezeichnet, die sich in diesem Sichtbereich befinden. Zusätzlich zu den Hindernissen werden auch Schätze und die Homebase angezeigt. Mit Hilfe der Schieberegler lässt es sich auf der Karte navigieren. Der Ausschnitt, der angezeigt wird, verändert sich dementsprechend. Um eine bessere Orientierung zu erhalten wird zusätzlich in der Statuszeile die Koordinaten der Mouseposition angezeigt und der zusehne

Ausschnitt als grünes Rechteck in der Minimap.
Zu Anfang tauchten Probleme beim Rendering auf. Das Bild hat beim neu zeichnen geflackert. Dies wurde dadurch gelöst, dass der angezeigte Ausschnitt zuerst als ein Bild im Hintergrund erstellt wird und anschließend in den Vordergrund geschaltet wird. Die Bigmap dient nicht nur dazu Hindernisse und Schätze anzuzeigen, der Benutzer selbst kann neu Hindernisse und Schätze einfügen, indem er auf der Optionspalette einen entsprechenden Zeichenmodus auswählt und mit klicken und ziehen der Mouse über der Bigmap Hindernisse einzeichnet oder löscht.

6.1.2. Die Minimap

Die Minimap ist ein JPanel welches die gesamte Karte/ den gesamten Quadtree anzeigt. Wegen Renderingproblemen die am Anfang aufgetaucht sind, wird die Karte jetzt zuerst als Bild im Hintergrund gezeichnet. Das Bild hat dabei die Größe der Karte (Höhe x Breite) in Pixel. Anschließend wird das Bild skaliert und in den Vordergrund geschaltet. Würde man jedes Hindernisse für sich skalieren und dann zeichnen, würde es zu Darstellungsproblemen kommen. Die rechteckigen Hindernisse hätten keine glatten Ränder sondern an manchen Stellen Zacken.
Außer der Anzeige der gesamten Karte liefert die Minimap zusätzliche Informationen über den Ausschnitt der in der Bigmap angezeigt wird. Weil mit float Werten und int Werten hantiert wird, stimmt der angezeigte Ausschnitt der Bigmap mit der Information in der Minimap nicht immer. Es liegen leichte Abweichungen aufgrund der Rundung der float Werte vor.

6.1.3. Die Optionspalette

Die Optionspalette ist ein JPanel, welches mit dem GridBagLayout sämtliche Buttons in eine Ordnung bringt. Mit den Buttons könne verschiedene Kombinationen aktiviert werden. Buttons die grau markiert sind, sind aktiviert. Insgesamt gibt es zehn verschiedene Kombinationen.

1. **Drawing slowly small obstacles:** Es werden einzelne kleine rechteckige Hindernisse gezeichnet. Dafür werden der Draw-Button, der Normal-Button und der „kleine Rechtecke“-Button aktiviert.

2. **Deleting slowly small obstacles:** Es werden einzelne kleine rechteckige Hindernisse gelöscht. Dafür werden der Delete-Button, der Normal-Button und der „kleine Rechtecke“-Button aktiviert.

3. **Drawing big obstacles:** Es werden große rechteckige Hindernisse gezeichnet. Dabei wird beim ziehen der Mouse mit gedrückter Mousetaste die Größe des zu zeichnenden Hindernisses festgelegt. Dafür werden der Draw-Button, der Normal-Button und der „große Rechtecke“-Button aktiviert.

4. **Deleting big obstacles:** Es werden große rechteckige Hindernisse gelöscht. Dabei wird beim ziehen der Mouse mit gedrückter Mousetaste die Größe

des zu löschenden Bereichs definiert. Dafür werden der Delete-Button, der Normal-Button und der g̈roße Rechtecke¨-Button aktiviert.

5. **Drawing treasures:** Es werden Schätze platziert. Dafür werden der Draw-Button, der Normal-Button und der Treasure-Button aktiviert.

6. **Deleting treasures:** Es werden Schätze entfernt. Dafür werden der Delete-Button, der Normal-Button und der Treasure-Button aktiviert.

7. **Setting the homebase:** Die Homebase wird platziert. Dafür werden der Draw-Button, der Normal-Button und der Homebase-Button aktiviert.

8. **Deleting the homebase:** Die Homebase wird entfernt. Dafür werden der Delete-Button, der Normal-Button und der Homebase-Button aktiviert.

9. **Drawing fast small obstacles:** Es werden viele kleine Hindernisse bei gedrückter Mousetaste hintereinander gezeichnet, vergleichbar mit einem Stift. Dafür werden der Draw-Button, der Fast-Button und der k̈leine Rechtecke¨-Button aktiviert.

10. **Deleting fast small obstacles:** Es werden viele kleine Hindernisse bei gedrückter Mousetaste hintereinander gelöscht, vergleichbar mit einem Radiergummi. Dafür werden der Draw-Button, der Fast-Button und der k̈leine Rechtecke¨-Button aktiviert.

Zusätzlich zu den Buttons, befindet sich ein Spinner auf der Optionspalette. Der Spinner ist deaktiviert, wenn keine Schätze gezeichnet werden. Wenn ein Schatz gezeichnet werden soll, wird der Spinner automatisch aktiviert so dass vor dem Zeichnen des Schatzes die Gröse des Schatzes eingestellt werden kann. Der Default-Wert liegt bei 100.

6.1.4. Die Menüleiste

Das Menü besteht aus vier Untermenüs: Dem Filemenü, dem Editmenü, dem Extrasmenü und dem Helpmenü.
Aus dem Filemenü heraus kann ein neues Terrain erstellt werden, dazu wird ein Diaolgfeld geöffnet, in dem man die Gröse der Karte in dem Spinner angeben muss. Des Weitern können Terrains und Schätze aus *.xml-Dateien geladen werden oder gespeichert. Hierbei wird auch jeweils ein Dialogfeld geöffnet in dem man den entsprechenden Pfad der zuladenden oder speicherten Datei angeben muss.
Aus dem Editmenü können alle Hindernisse von der Karte gelöscht werden, dafür wird der komplette Quadtree gelöscht bis auf den Wurzelknoten. Es könne auch alle Schätze entfernt werden, dafür werden alle Elemente aus der Liste gelöscht. Es ist auch möglich den Zoomfaktor der Bigmap zu Ändern, dafür öffnet sich ein Dialogfeld, der einen Spinner und zwei Buttons enthält. Beim Speichern des Faktors wird die Ansicht in der Bigmap dementsprechend sofort aktualisiert.
Aus dem Extrasmenü kann der Generator gestartet werden, die Gröse der

Schätze verwaltet oder eine Roboterkonfiguration erstellt werden.
Aus dem Helpmenü heraus kann nur die Bedienungsanleitung des Editors geöffnet werden.

6.1.5. Die Statuszeile

Die Statuszeile dient zur Orientierung des Benutzers, so zeigt sie z.B. die Koordinaten der Mouseposition auf der Karte, die in der Bigmap dargestellt wird an. Des Weiteren weiß der Benutzer wie groß die Karte ist und um welchen Faktor der Ausschnitt der Karte in der Bigmap vergrößert ist.

6.2. Schätzeverwaltung

Die Verwaltung der Schätze geschieht in einem separaten Dialogfeld, welches aus der Menüleiste heraus geöffnet wird. Das Dialogfeld ist aus einer Tabelle und zwei Buttons aufgebaut (Abbildung 48). Die Tabelle enthält zwei Spalten, wobei nur die Zellen der zweiten Spalte vom Benutzer bearbeitet werden können. Jede Zelle der zweiten Spalte enthält einen Spinner mit dem die Größe eines Schatzes festgelegt wird. Die erste Spalte enthält lediglich die Koordinaten eines Schatzes, der vorher in die Karte eingezeichnet wurde. Mit dem Save-Button werden

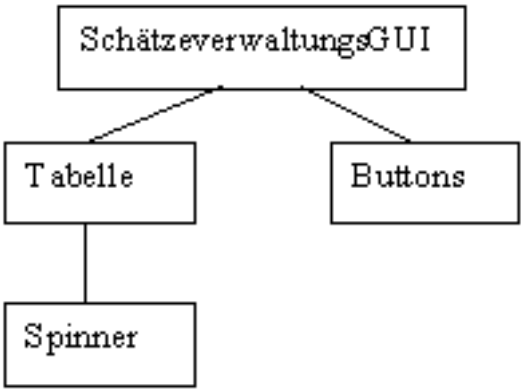


Abbildung 48: Grober Aufbau der SchätzeverwaltungsGUI

die Größen der Schätze intern gespeichert, es wird zu diesem Zeitpunkt keine *.xml-Datei erstellt. Die Schätze sind in einer Liste gespeichert, beim Speichern der Änderungen wird die Liste durchgelaufen und die bisherigen Werte durch die Werte in den jeweiligen Spinnern ersetzt. Mit dem Cancel-Button werden die Änderungen der Schatzgrößen nicht gespeichert. Das Dialogfeld wird nach der Betätigung jedes Buttons geschlossen.

6.3. Roboterkonfiguration

Die Roboterkonfiguration wird ebenfalls in einem separaten Dialogfeld erstellt (Abbildung 49). Das Dialogfeld ist in drei Bereiche unterteilt. Der obere Be-

reich enthält vier Buttons, mit denen eine neue Roboterkonfiguration erstellt (New-Button), eine bereits bestehende Roboterkonfiguration zu der in Bearbeitung befindlichen Roboterkonfiguration hinzugefügt (Add-Button), eine Roboterkonfiguration geladen (Load-Button) oder die aktuelle gespeichert werden kann (Save-Button).
Der mittlere Bereich enthält eine Tabelle mit 14 Spalten, von denen die erste Spalte nie bearbeitet werden kann sowie alle grauen Zellen. Die Tabelle kann beliebig viele Zeilen haben, wobei jede Zeile für einen Robotertyp steht und jede Zelle der Zeile die Eigenschaften für diesen Robotertyp festlegt. Die Zellen der zweiten bis 12 Spalte enthalten alle eine Spinner, wobei einige Spinner mit reellen Werten arbeiten und andere nur mit natürlichen Werten. Die Zellen der 13. Spalte enthalten Textfelder in den der Pfad der zu verwendeten Strategie eingetragen ist. Die Zellen der letzten Spalten enthalten je einen Button, mit dem ein Dialogfeld geöffnet wird. Mit diesem lässt sich die zu verwendete Strategie auswählen. Der Pfad der Strategie wird anschließend in der 13. Spalte angezeigt.
Der untere Bereich enthält wiederum Buttons, sechs an der Zahl. Mit den ersten fünf Buttons lässt sich jeweils eine Zeile in die Tabelle einfügen, dabei steht jeder Button für einen Robotertyp. Beim Einfügen einer Zeile wird dann auch gleich darauf geachtet, dass entsprechenden Zellen grau markiert sind, d.h. nicht bearbeitbar sind, da der Robotertyp diese Eigenschaften nicht hat oder nicht benötigt.. Der letzte Button dient zum löschen einer Zeile. Es kann jede beliebige Zeile gelöscht werden bis auf die erste, die die Informationen der Homebase enthält.

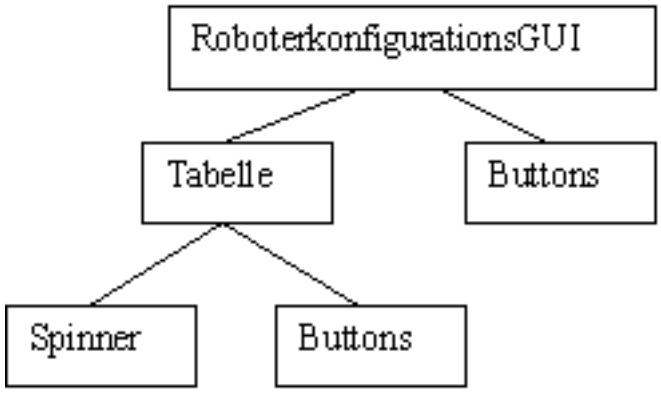


Abbildung 49: Aufbaue der RoboterkonfiguratiosnGUI

Beim Neuerstellen einer Roboterkonfigarution (New-Button betätigen) werden alle Zeile aus der Tabelle gelöscht und anschließend die Zeile, die die Informationen der Homebase enthält mit Defaultwerten eingefügt.
Beim Einfügen einer anderen Roboterkonfiguration zu der aktuellen (Add-Button betätigt), wird die entsprechende *.xml-Datei geöffnet, die Informationen daraus gelesen und entsprechend als Zeilen in die Tabelle eingefügt. Die

Eigenschaften der Homebase ändern sich nicht.
Beim Laden einer Roboterkonfiguration (Load-Button betätigt) werden alle Zeilen der aktuellen Tabelle komplett gelöscht. Dann werden alle Informationen aus der ausgewählten *.xml-Datei gelesen und entsprechend als Zeilen in die Tabelle eingefügt. Die Eigenschaften der Homebase ändern sich auch, wobei die Position nicht geändert werden kann. Damit wird verhindert, dass die Homebase in einem Hindernis steht.
Beim Speichern der Roboterkonfiguration (Save-Button betätigt) wird eine neue *.xml-Datei mit den Daten aus der Tabelle erstellt. Die Informationen der Tabelle werden Zeilenweise ausgelesen.

6.4. Datenstruktur

Der Editor benutzt zwei verschiedene Datenstrukturen:

1. Liste: Alle Schätze, die auf der Karte platziert werden, werden in einer LinkedList gespeichert.

2. Quadtree: Alle Hindernisse, die auf der Karte platziert werden, werden in einem RegionQuadtree gespeichert. Der Quadtree ist folgendermassen aufgebaut: Er enthält einen Wurzelknoten und jeder Vaterknoten bis zu vier Kindsknoten. Jeder Blattknoten steht für ein Hindernis auf der Karte. Jeder Knoten steht für einen Bereich auf der Karte. Der Wurzelknoten enthält die Information der gesamten Karte (Abbildung 50).

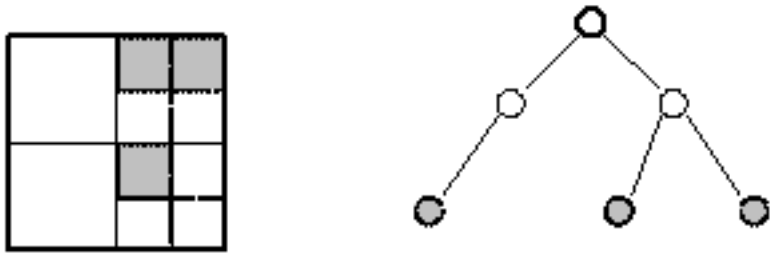


Abbildung 50: Aufbau eines RegionQuadtrees

Die Karte lässt sich aufteilen in vier kleinere Bereiche, den nordwestlichen, nordöstlichen, südwestlichen und südöstlichen Bereich. Diese Bereiche bilden die Kindknoten des Wurzelknotens, dies lässt sich rekursiv weiter verkleinern. Ein Knoten enthält nur dann einen Kindknoten, wenn sich in diesem Bereich ein Hindernis befindet. Die Karte wird auf diese Weise soweit in Bereiche unterteilt, bis ein Bereich so klein wie das kleinste Hindernis ist. Befinden sich in allen vier Blattknoten ein Hindernis, so wird der Bereich zu dem nächst grösseren zusammengefasst, d.h. Die Blattknoten werden gelöscht und der Vaterknoten wird zum Blattknoten, dies spart Speicherplatz.

6.5. Generator

Da viele Strategien eine Menge von Karten brauchen, wurde ein Werkzeuge benötigt, das die Generierung von Karten automatisch übernimmt. Zu diesem Zweck wurde ein Generator entwickelt, der ein Bestandteil des Editors ist und zwei Methoden „Random function“und „File function“zur Generierung einer Karte bietet. Der Generator wurde in Form eines Wizards implementiert.

6.5.1. Architektur

In Abbildung 51 sind die wesentlichen Klassen des Generators dargestellt. Er besteht aus 3 Komponenten: eigentliche Implementierung des Generators, Wizard-Framework (Package „wizard“) und Hilfsklassen für „Random function“(Package „forms“). Alle Eingaben, die ein Benutzer im Generator eingibt, werden in der

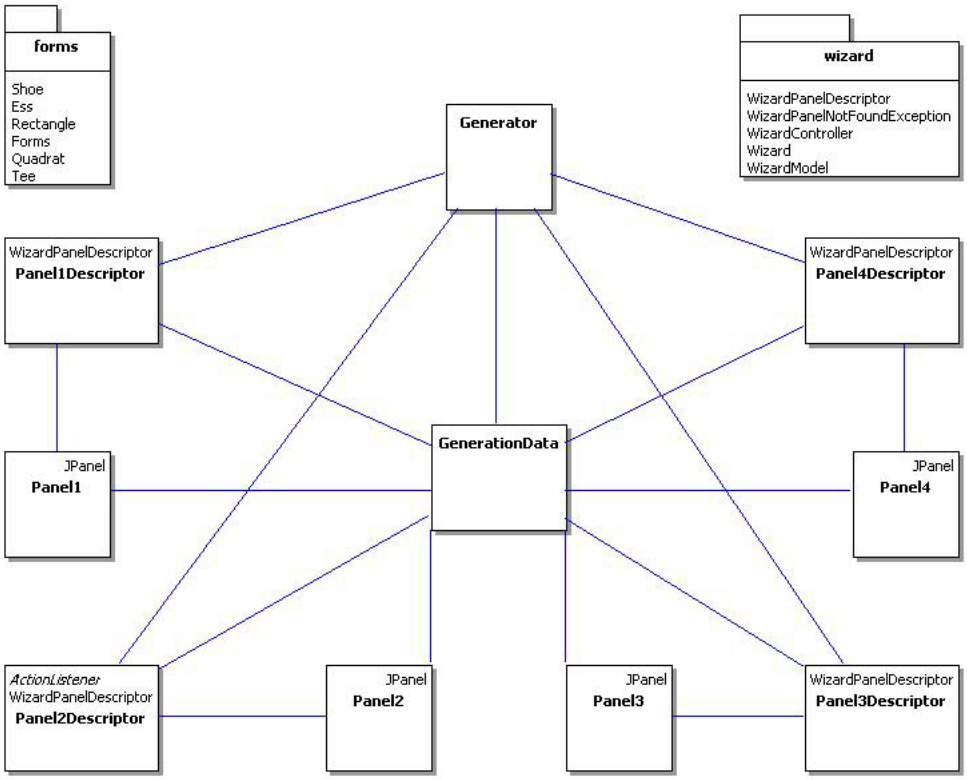


Abbildung 51: Klassen des Generators

Klasse „GenerationData“gesammelt und bei der Generierung einer Karte ausgelesen. Die genauen Funktionen jeder einzelnen Panel werden im Kapitel 6.5.3 erklärt.

6.5.2. Wizard Framework

Für die Implementierung des Generators wurde ein Framework benutzt, dessen Klassen in Abbildung 52 dargestellt sind. Das Framework bietet die Möglichkeit,

einen Wizard mit „Back“, „Next“und „Cancel“-Buttons komfortabel zu erstellen.

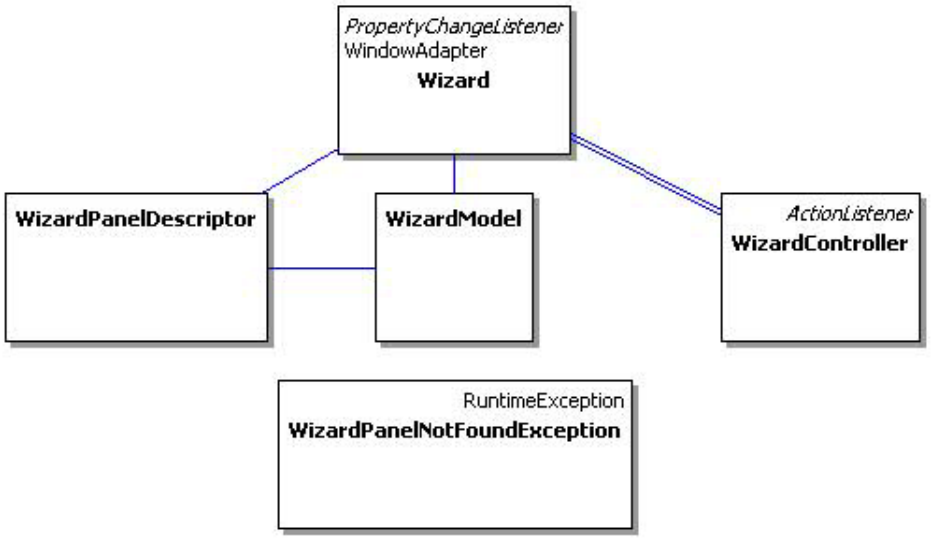


Abbildung 52: Klassen des Wizards

Die Klasse **Wizard**
Diese Klasse implementiert einen grundlegenden Assistenten-Dialog, wo Programmierer eine oder mehrere Komponenten wie Panels behandeln können. Diese Panels können beliebig durch Verwendung von „Back“oder „Next“Buttons navigiert werden, oder den Dialog schließen, mit dem „Cancel“angeklickt wird. Beachten Sie, dass, obwohl der Dialog ein CardLayout Manager verwendet, ist die Reihenfolge der Felder nicht linear. Jede Gruppe bestimmt, während der Laufzeit, was ihre nächsten und vorherigen Panels sind.

Die Klasse **WizardController**
Diese Klasse behandelt alle Ereignisse, die durch Drücken eines der drei Buttons „Next“, „Back“und „Cancel“erzeugt werden. In Abhängigkeit, was für ein Button gedrückt wurde, zeigt der Controller das aktuelle Panel und setzt die Zustände von Buttons.

Die Klasse **WizardModel**
Das Model für die Wizard Komponente, enthält Text, Symbole und aktiviert einzelne Tasten, sowie aktuelle Seite, die angezeigt wird. Beachten Sie, dass das Modell, in seiner jetzigen Form, nicht eine Subklasse ist.

Die Klasse **WizardPanelDescriptor**
Diese Klasse wird als Referenz für ein Panel im Wizard benutzt, und beinhaltet alle Regeln, wie sich ein Panel verhalten soll.

6.5.3. Generatorimplementierung

Der Generator-Wizard besteht aus 4 Seiten:

- „Main settings“(Panel1 sieh Abb. 51)
- „Settings for random function“(Panel2 sieh Abb. 51)
- „Settings for file function“(Panel4 sieh Abb. 51)
- „Creating the map“(Panel3 sieh Abb. 51)

Das Panel **„Main settings“**
Hier werden alle grundlegenden Informationen zur Generierung der Karte abgefragt.

- Grösse der Karte als Potenz von 2 (die Größe der Karte ist dann 2^x)
- Anteil von Hindernissen an der gesamten Karte, also wieviel von der gesamten Fläche mit Hindernissen bedeckt werden muss.
- Die Generierungsfunktion. Zur Zeit gibt es nur zwei Methoden: „Zufallsfunktion“und „Dateifunktion“

Das Panel **„Settings for random function“**
Diese Methode bietet eine Möglichkeit, eine Karte durch einfügen von verschiedenen Formen zu generieren. Zur Zeit gibt es folgende Formen:

- „Quadrat“- ein Rechteck
- „Rectangle“- ein Recheck
- „Shoe“- eine Hufeisenform
- „Tee“- eine T-Form
- „Ess“- eine S-Form

Alle Formen sind standardmässig ausgewählt. Falls nur bestimmte Formen bei der Generierung benutzt werden müssen, kann man die unnötige Formen abwählen.
Zu jeder Form kann man die minimal- und maximalmögliche Grösse ausgewählen. Bei der Generierung der Karte werden dann diese Formen aus dem Bereich „minimale“und „maximale“Grössen per Zufall erzeugt und in die Karte eingefügt. Zusätzlich gibt es auch eine Möglichkeit, den Anteil einer Form an dem Gesamtanteil aller Formen einzugeben. Die Berechnung erfolgt folgendermassen:

$$\texttt{Anteil einer Form von 100} = \frac{\texttt{Eingegebener Wert dieser Form}}{\texttt{Summe aller Werte von ausgewählten Formen}}$$

In Abbildung 53 sind alle Klassen, die „Random function“benutzt, dargestellt.

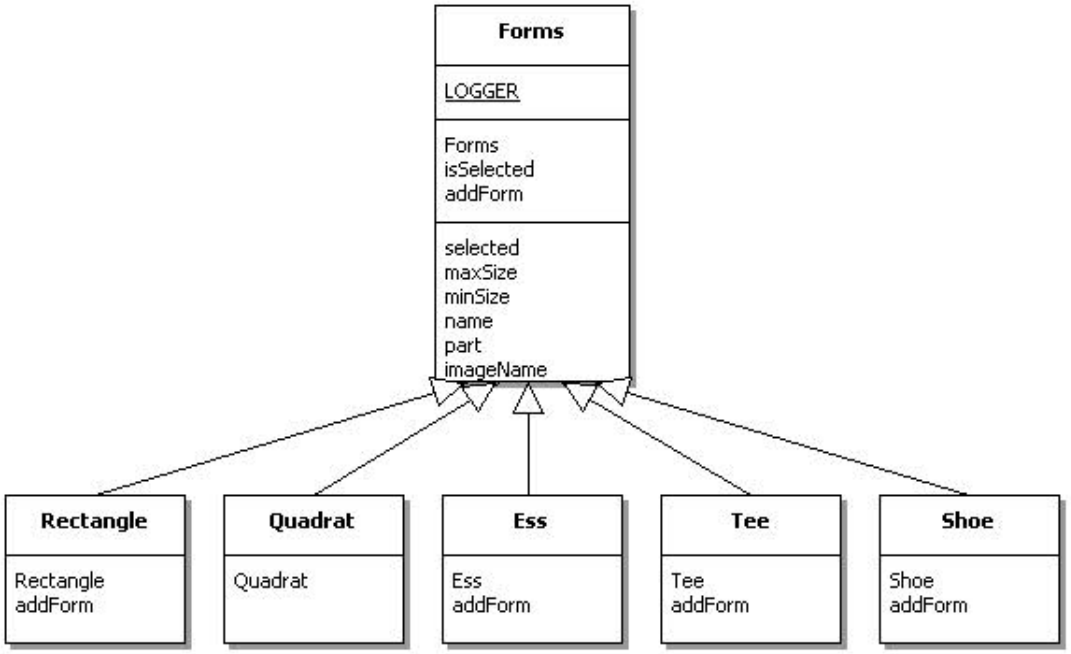


Abbildung 53: Hilfsklassen für „Random function“

Das Panel **„Settings for file function“**
Hier können Parameter für „File function“eingestellt werden. Diese Funktion dient der Generierung einer Karte aus einem Bild. Zur Zeit werden BMP, JPG, GIF und PNG Bild-Formate unterstützt. Bei der Generierung wird alles, was nicht **weiß** ist, als Hindernis interpretiert.

Das Panel **„Creating the map“**
Hier wird den Fortschritt der Generierung mit Hilfe eines Balken angezeigt. Bitte beachten Sie, dass die Generierung einige Zeit in Anspruch nehmen kann.

7. Messgrößen

7.1. Einleitung

Ein Hauptziel der PG-Smarteams war es eine Plattform zu entwickeln, um unterschiedliche Strategien in der Simulationsumgebung durchspielen zu können. Um diese Strategien besser zu analysieren wurde ein *Framework* entwickelt, um relevante Daten der Strategien zu aggregieren und zu visualisieren.
Dabei wurde besonders Wert auf leichte Benutzung und Flexibilität des *Frameworks* gelegt.
Da je nach Problemstellung und deren Lösung verschiedene Messgrößen innerhalb unterschiedlicher Strategien interessant sind, bietet das *Framework* die Möglichkeit benutzereigene Messgrößen zu sammeln und in Diagrammen anzuzeigen[1]. Im folgenden Abschnitten werden Einzelheiten zur Implementierung präsentiert.

7.2. Ablauf und Architektur

Der Kern des Messgrößenframeworks ist die Klasse *MeasurmentContolGUI*. Dort werden die relevanten Daten gesammelt und bei Bedarf angezeigt. Die GUI-Elemente *(Tabelle, Button usw.)* und deren Reaktionen sind ebenfalls in dieser Klasse untergebracht. Die Klasse *MeasurmentContolGUI* ist ein Obser-

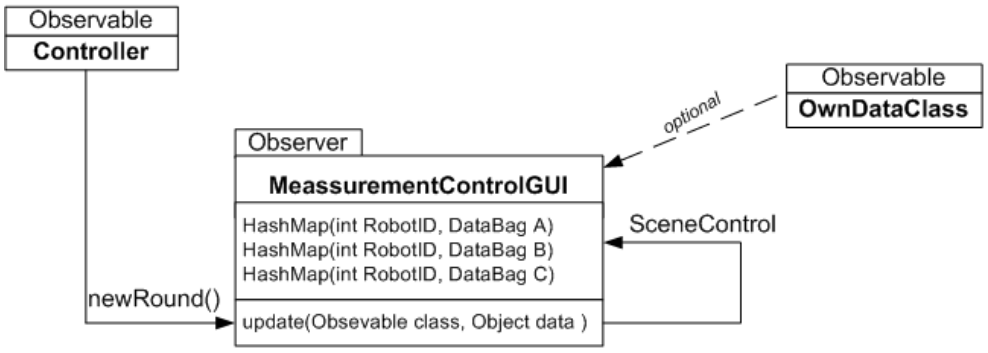


Abbildung 54: Architektur des Messgrößen *frame-works*.

ver [2], den man in 'observable' Klassen registrieren und deregistrieren kann.

Sobald der Benutzer auf den Messgrößen-Button in der Kernel-GUI klickt, wird eine Instanz von *MeasurmentContolGUI* erzeugt und im *Controller* registriert. Ab diesem Zeitpunkt wird vom Controller die aktuelle Rundenzahl an die Instanz von *MeasurmentContolGUI* verschickt. Die Rundenzahl landet in der **update()**-Methode.
Mit dem Spinner **update()**-Intervall in der GUI kann man die Häufigkeit der Aktualisierung festlegen. Dabei wird die aktuelle Rundenzahl *modulo* update-Intervall gerechnet. Setzen wir einfachheitshalber das **update()**-Intervall auf

[1] Das ist mit Flexibilität gemeint.
[2] In der Literatur manchmal auch als Listener bezeichnet

10 d.h. jede zehnte Runde können wir eine Aktion durchführen.

Ziel dieser Vorgehensweise ist es jede x-te Runde Daten aus den Datenstrukturen herauszuholen und in eigenen Datenbehältern (*DataBags*) zu speichern, um sie bei Bedarf Diagrammen zu übergeben, die sie dann anzeigen können.

Der Zugriff auf die Daten erfolgt über die Klasse *SceneControl*. Dies hat den Vorteil, dass man auf alle Datenstrukturen des Kernels (*Roboter, Geländespeicherung, dynamische Objekte*[1]) zugreifen kann. Es ist somit auch möglich auf einzelne Roboter und deren Datenbehälter *(StandardDetails(), ExploredMap, ElementaryAction usw.)* zu zugreifen und daraus die relevanten Daten heraus zu ziehen.

7.3. Sammeln der Messwerte

Wie oben beschrieben können jede x-te Runde Daten herausgezogen, bearbeitet und in eigenen Datenbehältern gespeichert werden. Die Daten können je nach Wahl des Diagrammtyps in zwei Behältern gespeichert werden:

1. DefaultCategoryDataset dataset *(Ein und zwei Flächendiagramm)*
2. DefaultPieDataset pieDataset *(Kuchendiagramm)*

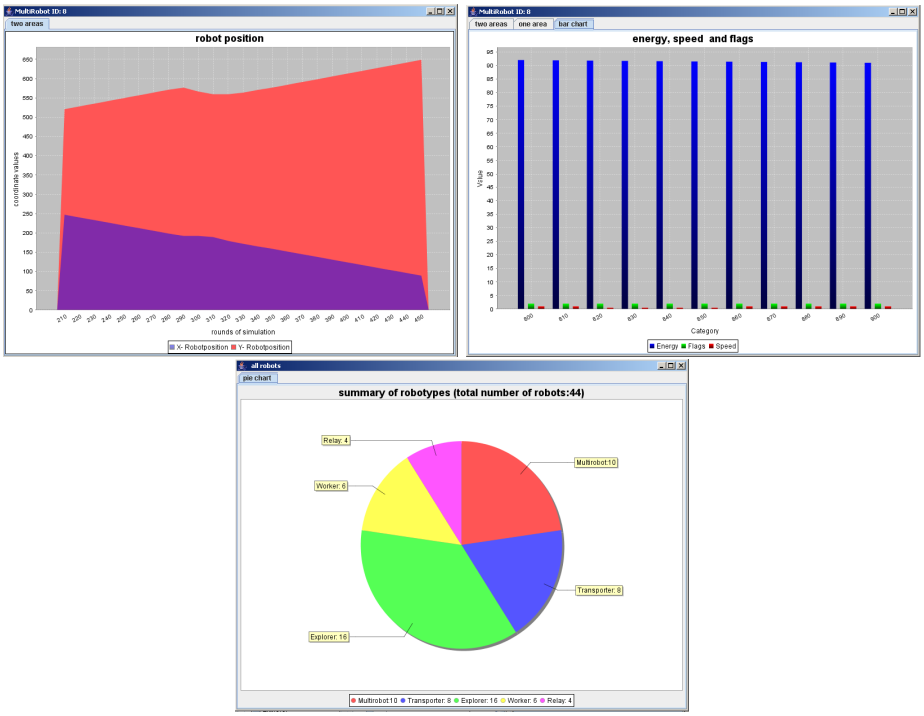


Abbildung 55: Zwei Flächendiagramm, Balken- und Kuchendiagramm.

Jede Aktion *(add, remove)* auf den Datenmengen wird unmittelbar im Diagramm angezeigt. Bei längeren Simulationen oder kurzen update-Intervallen

[1] Schätze und Flags.

werden die Werte auf der x-Achse gestaut. Wenn *auto remove* in der GUI aktiviert wird, werden Staus erkannt und jeweils das erste Element gelöscht um Platz für das ankommende Element zu schaffen.

7.4. Benutzereigene Messgrößen

Um eigene Messgrößen zu erfassen muss man erst überlegen von welchen Datenstrukturen man sie sammeln möchte. Der einfachste Weg wäre es die bereits bestehende Datenstruktur vom Kernel- und Roboter-Modul zu benutzen und sie gegebenenfalls um eigene Parameter zu erweitern.

Anhand eines Beispiels möchte ich demonstrieren wie man vorgehen würde:
Wir möchten anzeigen wie viel Prozent der Karte von welchen Robotertypen erforscht wurden [1]. Dazu wählen wir das Kuchendiagramm, um die Ergebnisse anzuzeigen. Als erstes müssen wir entscheiden woher wir die Daten bekommen. Eine einfache Möglichkeit wäre es über *SceneControl* auf einen Roboter zuzugreifen *IKernelToRobot* → *RegionQuadTree*. Damit hätte man aber nur einen Roboter und nur einen Teil der erkundeten Karte.
Bei der Homebase werden alle erkundeten Kartenausschnitte zusammengetragen, um festzustellen ob die gesamte Karte exploriert wurde. Da könnte man ansetzen und beim mergen der Karten an der Homebase jeweils den Robotertyp mit protokollieren (*HashMap*(**RobotType** *robot*, **int** *prozent*)). Man könnte dann anhand der RoboterID die Homebase aus der Roboter-Datenstruktur heraussuchen und die HashMap jede x-te Runde auslesen, um die Werte im Kuchendigramm zu aktualisieren.

Im Prinzip ist jede neue Messgröße mit Erweiterungen in bereits bestehenden Datenstrukturen möglich.

Optional kann man eigene Datenstrukturen einführen und völlig unabhängig von bereits bestehenden Strukturen Daten sammeln. Die Datenbehälter müssen nur 'observable' sein (siehe Abbildung 7.2), um *MeasurmentContolGUI* registrieren zu können.

[1] Explorer→60%, MultiRobot→20%, Transporter→9%, usw.

8. Ausblick/Offene Punkte

Die beschriebene Funktionalität unserer Software wurde zwar umgesetzt, doch einige Teile sind immernoch in einem ausbaufähigen Zustand, wobei hier als Beispiel das Teammanagement dienen könnte. Eine Erweiterungsmöglichkeit würde sich insoweit bieten, daß statt der bisherigen Zufallsentscheidungen konkrete Heuristiken implementiert werden können. Solche "Baustellen" sollte man en masse finden können. Bis jetzt sind unsere Datenstrukturen und die damit verbundenen Algorithmen in der Lage, eine Simulation mit 1000 Robotern auf Karten, vertreten durch einen Quadtree mit 1,3 Millionen Blättern, zu ermöglichen. Dennoch können einige Algorithmen sowie einige Datenstrukturen verbessert werden, um die Leistungsfähigkeit des Simulators zu verbessern. Man könnte zum Beispiel eine Parallelisierung des Systems in Erwägung ziehen, um die Möglichkeit zu geben die Simulation auf mehrere Rechner zu verteilen, oder einzelne Roboter in Threads auszulagern. Ein anderer Ansatz ist die Entwicklung und Erprobung von Strategien und Taktiken für die Roboter, wobei dem Benutzer auch ein elegantes Framework zur Verfügung steht. Interessant wären zum Beispiel Algorithmen, die in der Lage sind, gezielt Hindernisse zu umrunden (und das auch ermitteln zu können), oder eine Strategie, die die vollständige Erkundung einer Karte garantieren kann. Genauso könnte man auch versuchen die grafischen Features aus dem 2D ins 3D zu übertragen und es bieten sich (fast unendlich) viele neue Möglichkeiten der Visualisierung von Details der Roboterstrategie (zukünftig geplante Wegpunkte, Nachrichtenverkehr, Roboter eines Teams, ...). Natürlich eröffnen neue Taktiken und Strategien auch neue Möglichkeiten, wichtige Daten daraus in den Messgrößen zu visualisieren.
Den Simulator und weitere Informationen erhalten Sie auf der Website
`http://www.upb.de/cs/smartteams`

A. Projekt in Eclipse einbinden

Soll das Projekt aus dem CVS in Eclipse eingebunden werden, so sind ein Paar Dinge zu beachten. Daher hier eine kurze Anleitung:

Projekt aus CVS auschecken Der Code ist als vollständiges Eclipse-Projekt im CVS enthalten. Man kann es in Eclipse einbinden, indem man beim Anlegen eines neuen Projekts "project from CVS" auswählt. Die notwendigen Einstellungen sind (zum Zeitpunkt des Schreibens dieses Abschnittes):

- host: sshgate.cs.upb.de
- path: /home-f/fg-madh/pg-smartteams/
- user: IMT-Userdaten
- connection type: extssh
- module: smartteam

Da die JOGL-Bibliotheken plattformabhängig sind, muss jeder Benutzer individuell einstellen, welche Version zu verwenden ist:

- in der Menüleiste "Window → Preferences" anwählen
- auf der linken Seite "Java → Build Path → User Libraries" auswählen
- ganz rechts per "New..." die User Library "jogl" erstellen
- den neu erstellten Eintrag "jogl" auswählen
- dann mit "Add JARs..." folgende Bibliothek hinzufügen: "[Pfad zum Projekt]/lib/[Betriebssystem]/jogl.jar"
- nochmal den Eintrag "jogl" auswählen
- die "gluegen-rt.jar" aus dem selben Ordner hinzufügen
- beide JAR-Einträge aufklappen, dort "Native library location" auswählen
- auf der rechten Seite per "Edit..." folgenden Pfad im Workspace einstellen: "smartteam/lib"/[Betriebssystem]/" einstellen

Jetzt sollte der Code alle Bibliotheken finden und kompilierbar sein. Um das Projekt zu starten muss in der Start-Konfiguration unter dem Punkt "Arguments → VM Arguments" folgender Eintrag gemacht werden:

"-Djava.security.policy=$project_loc/security.policy"

sonst fliegen Exceptions wegen des RMIs. Weiterhin ist es empfehlenswert, mehr RAM einzustellen mittels: "-Xmx1G".

Literatur

[3D] Lighthouse 3D. Glsl tutorial. http://www.lighthouse3d.com/opengl/glsl/.

[Fis07] Matthias Fischer. *Vorlesung "Algorithmen in der Computergrafik" – Räumliche Datenstrukturen.* SS 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, Boston, MA, January 1995.

[Hol] Vic Hollis. 3d lens flare with occlusion testing. http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=44.

[Hun] Jason Hunter. Jdom. http://www.jdom.org.

[KKF+04] Jan Klein, Jens Krokowski, Matthias Fischer, Michael Wand, Rolf Wanka, and Friedhelm Meyer auf der Heide. The randomized sample tree: A data structure for interactive walk-throughs in externally stored virtual environments. *PRESENCE*, 13(6):617–637, December 2004. The MIT Press.

[Mor] Mark Morley. Frustum culling in opengl. http://www.crownandcutlass.com/features/technicaldetails/frustum.html.

[Sam90] H. Samet. *Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[Tha00] Ulrich Thatcher. *Game Programming Gems*, chapter Loose Octrees. Charles River Media, 2000.

Abbildungsverzeichnis

1. Illustration des Modells . . . 2
2. Die Architektur des Gesamtsystems . . . 5
3. Die Architektur des Simulationskernels . . . 7
4. Protokoll bei Verfügbarkeit der Karte auf Visualisierungsseite . . . 12
5. Protokoll bei fehlender Karte auf Visualisierungsseite . . . 13
6. Beispielaufteilung eines Terrain in Zellen . . . 14
7. Beispiel für einen RegionQuadtree . . . 17
8. Quadtree mit Nachbarschaftsbeziehung vor Zellteilung . . . 18
9. Quadtree mit Nachbarschaftsbeziehung nach Zellteilung . . . 19
10. Aufbau eines PRQuadtrees . . . 20
11. Klassenübersicht . . . 20
12. Aufbau *(links)* und Aktionszyklus des Roboters *(rechts)* . . . 22
13. Wesentliche Klassen des Roboter-Moduls. . . . 23
14. Vermeidung von Kollisionen mit Hindernissen *(links)* und Robotern *(rechts)* . . . 25
15. Funktionsweise der StaticMap . . . 26
16. DynamicDataList . . . 27
17. UML-Klassendiagramm des Taktik-Konzeptes . . . 32
18. `SidestepRobotEvasion` mit Parameterwert “AngularRight” lässt Roboter schräg aneinander vorbeigehen. . . . 42
19. Exemplarisches Vorgehen der Taktik `WalkAlongBoundaries` . . . 43
20. Protokoll beim Teamaufbau . . . 48
21. Die Auktion bei der Teambildung . . . 49
22. Protokoll beim Verlassen eines Teams . . . 50
23. Protokoll beim Verlassen eines Teams . . . 51
24. Zustandsautomat der Strategie *SubAreaStrategy* . . . 56
25. Beispiel einer SubArea mit Stripes . . . 57
26. Beispiel zum Ablaufen eines Hindernisses bis zum Rand der SubArea . . . 57
27. Beispiel zum Umrunden eines Hindernisses . . . 58
28. Beispiel zum Verlassen der SubArea und Erkunden von abgetrennten Bereichen . . . 58
29. Zustandsautomat der Strategie TreasuregreedStrategy . . . 61
30. Zustandsautomat der Worker-Strategie . . . 62
31. Zustamdsautomat von der Transporter-Strategie . . . 63
32. Die Visualisierung . . . 64
33. Grobarchitektur der Visualisierung . . . 65
34. Repräsentation der Karte im TerrainQuatree in unterschiedlichen Detailgraden. In gröberen Detailgraden werden die Werte der Kinder interpoliert, um auch feine Strukturen bei kleinen Zoomstufen angenähert darstellen zu können. . . . 69
35. Resultat des Geländezeichnens ohne und mit Interpolation. . . . 69
36. Die Robotertypen: Relais, Explorer, Worker, Multirobot und Transporter (von links nach rechts). . . . 72
37. Die Blickrichtung der Kamera . . . 72

38. Zusammenhängende Hindernisstrukturen finden 75
39. Hindernisstrukturen an den Ecken höher auflösen 75
40. Quadtree balancieren 75
41. Iterativ Abstände zum Rand berechnen 76
42. Dreiecke generieren 76
43. Einfügen eines kleines Dreiecks in einen Loose Octree 77
44. Beispiel für *pullUpSamples* 78
45. Beispiel für *compressTreeStructure* 79
46. Verschieben des Mittelpunktes bei *shrinkBoundingBoxes* 79
47. Grober Aufbau der GUI 81
48. Grober Aufbau der SchätzeverwaltungsGUI 84
49. Aufbaue der RoboterkonfiguratiosnGUI 85
50. Aufbau eines RegionQuadtrees 86
51. Klassen des Generators 87
52. Klassen des Wizards 88
53. Hilfsklassen für „Random function“ 90
54. Architektur des Messgrößen *frame-works.* 91
55. Zwei Flächendiagramm, Balken- und Kuchendiagramm. 92

Tabellenverzeichnis

1. Message-Typen Teil 1 10
2. Message-Typen Teil 2 11
3. Zustandsautomat der Klassse MainControl 66
4. Update-Methoden für die Ablaufsteuerung im Visualisierungsmodul 67
5. Namensunterschiede von Methoden zwischen Implementierung und Paper 77
6. Testergebniss der Simulation mit unterschiedlichen Karten 80